VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Reinforcement Learning pro stolní hry

Reinforcement Learning for Table-games

David Bohunovský

Diplomová práce

Vedoucí práce: doc. Ing. Jan Platoš, Ph.D.

Ostrava, 2021

Abstrakt

Cílem této diplomové práce je prostudování tématu zpětnovazebního učení a následného aplikování jej na vybrané stolní hry. Tato práce popisuje jednotlivé postupy strojového a zpětnovazebního učení. Dále jsou v práci popsány vybrané algoritmy, které byly naimplementovány a následně otestovány na vybraných stolních hrách. V závěru práce je vyhodnocení jednotlivých algoritmů v rámci jejich testování a shrnutí o tom, které algoritmy se prokázaly jako nejúspěšnější.

Klíčová slova

strojové učení, zpětnovazební učení, neuronová síť, stolní hry

Abstract

The aim of this master thesis is to study the topic of reinforcement learning and subsequent application of it to selected board games. This work describes the individual procedures of machine and reinforcement learning. Furthermore, selected algorithms are described in the work, which were implemented and subsequently tested on selected board games. At the end of the work is the evaluation of individual algorithms in their testing and a summary of which algorithms have proven to be the most successful.

Keywords

machine learning, reinforcement learning, neural network, board games

Poděkování

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Obsah

Seznam použitých symbolů a zkratek **6**

Seznam obrázků **7**

1 Úvod **9**

2 Machine Learning **10**
2.1 Učení s učitelem 10
2.2 Učení bez učitele 11
2.3 Využití machine learningu 11

3 Reinforcement Learning **12**
3.1 Markovův rozhodovací proces 13
3.2 Markovova properta 14
3.3 Markovův proces a markovův chain 14
3.4 Markovův reward proces 15
3.5 Výzvy v reinforcement learningu 16

4 Minimax Algoritmus **17**
4.1 Transposition table 19
4.2 Alpha beta pruning 20

5 Q-Learning **21**
5.1 Provedení a vykonání akce 22
5.2 Aktualizace Q-table 23

6 Deep Q-Learning **24**
6.1 Neuronové Sítě 24
6.2 DQN 27
6.3 Double DQN 28
6.4 Dueling DQN 29

7 Konvoluční sítě **31**
7.1 Convolutional layer 32
7.2 Pooling layer 33

8 Implementace **34**
8.1 Využité technologie 34
8.2 Architektura 34
8.3 Herní prostředí 35
8.4 Hráč 37
8.5 Zobrazení statistik a průběh testování 37
8.6 Minimax 40
8.7 Q-Learning 42
8.8 DQN 44
8.9 Konvoluční sítě 46

9 Testování **48**
9.1 Q-Learning 48
9.2 DQN 49

10 Závěr **57**

Literatura **58**

Seznam použitých zkratek a symbolů

RL	–	Reinforcement learning
ML	–	Machine learning
DQN	–	Deep Q-Network
MRP	–	Markov reward process
MDP	–	Markov decision process
NN	–	Neural network
CNN	–	Convolution neural network

Seznam obrázků

3.1 Model RL [4] . . . 12
3.2 Vyobrazení cart-pole problému [6] . . . 13
3.3 Markovův chain [7] . . . 14
3.4 Markova odměna [7] . . . 15

4.1 Aktuální stav herní plochy . . . 17
4.2 Nasimulované všechny možné tahy . . . 18
4.3 Ohodnocení všech finálních tahů . . . 18
4.4 Hodnoty po propagaci zpět do první vrstvy . . . 19
4.5 Ukázka výsledku s využitím alfa-beta ořezáváním . . . 20

5.1 Q Tabulka . . . 22

6.1 Neuronová Síť [16] . . . 25
6.2 Neuron [16] . . . 25
6.3 Ukázka aktivačních funkcí [18] . . . 26
6.4 DQN vs Dueling DQN [21] . . . 30

7.1 Schéma konvoluční sítě [22] . . . 31
7.2 Zpracování dat v konvolučních sítích . . . 32
7.3 Zpracování dat v konvolučních sítích . . . 32
7.4 Způsoby vybrání rysů [23] . . . 33

8.1 Ukázka grafu vytvořeného ze statistik testování . . . 40
8.2 Porovnání chování algoritmus . . . 41
8.3 Statistika z testu konvolučních sítí . . . 47

9.1 Testování QLearningu . . . 49
9.2 První test DQN vs Random . . . 50
9.3 Přepočet vstupních parametrů . . . 51
9.4 DQN a DoubleDQN . . . 52

9.5 Dueling DQN 52
9.6 Porovnání jednotlivých sítí 53
9.7 Dueling DQN vs QLearning před naučením 53
9.8 Dueling DQN vs Q-Learning po naučení 54
9.9 Dueling DQN vs QLearning po naučení 55
9.10 Testování DQN oproti Q-Learningu 56
9.11 DQN vs Q-Learning 56

Kapitola 1

Úvod

Cílem této diplomové práce bylo prozkoumat oblast strojového učení. Konkrétně se jednalo o oblast zvanou zpětnovazební učení, což je jedna z technik strojového učení. Tyto techniky měly být poté otestovány na vybraných stolních hrách. Mezi tyto hry byly zařazeny Piškvorky a Spoj 4. V práci budou spíše použity anglické názvy jako TicTacToe a Connect4.

V práci jsou podrobně popsány postupy strojového a zpětnovazebního učení. Dále jsou vysvětleny jednotlivé algoritmy, které byly vybrány k implementaci. Mezi tyto algoritmy patří minimax, který sice nepatří mezi algoritmy strojového učení, ale bude využit v pozdější fázi při testování. Dále byl vybrán algoritmus Q-Learning, jako první zástupce zpětnovazebního učení. Poté je podrobně rozebrán Deep Q-Learning, který využívá neuronové sítě a jeho následné zlepšování.

V druhé části práce bude rozebrána implementace těchto vybraných algoritmů. Dále bude popsaná implementace herního prostředí a dvou vybraných, již dříve zmíněných, zástupců stolních her. Také se podíváme na architekturu celého projektu neboli jak je vše spojeno do jedné funkční aplikace.

Závěrem práce bude porovnání těchto algoritmů ve vybraných stolních hrách. Jednotlivé algoritmy budou mezi sebou postaveny v určitém počtu zápasu. Po vyhodnocení všech zápasu je ze statistik vytvořen graf, který vynese výsledný počet výher, proher či remíz mezi algoritmy. Proti algoritmům se dá hrát také jako hráč. Jedna z možností testování je postavit se například Q-Learningu před naučením a poté znovu po naučení.

Kapitola 2

Machine Learning

Strojové učení je věda, která se zabývá programováním počítačů tak, aby byly schopné se samovolně učit z dat, které dostanou. Strojové učení se využívá téměř ve všech odvětvích běžného života jaký známe. Jeden z nejběžnějších příkladů je obyčejný spam filtr, který je obsažený v emailu. Programu předložíme takzvaná trénovací data. V tomto případě se jedná o množinu normálních emailů a spamů. Na základě těchto trénovacích dat se program dokáže naučit rozpoznat rozdíl mezi normálním emailem a spamem. Dalším velice běžným příkladem je doporučovací algoritmus, který se nachází například na Netflixu. Program zkoumá, jaké uživatel sleduje seriály či filmy a na základě těchto informací se vám snaží předložit film, který by mohl odpovídat vašim chutím.[1]

Strojové učení se hodí na problémy, které vyžadují obrovské množství ručního ladění. Algoritmus strojového učení nám velmi často pomůže dosáhnout lepších výsledků v kratším časovém úseku. Použití strojového učení může být vhodné v prostředích, ve kterých je obtížné využít klasické programovací techniky.

Strojové učení můžeme rozdělit do několika kategorií. Jedním z rozdělení je, zda program k naučení potřebuje, či nepotřebuje trénovací data. Učení bez použití trénovacích dat se nazývá zpětnovazební učení a bude probráno později. Učení, které potřebuje trénovací data, se dá dále rozdělit na učení s učitelem a učení bez učitele.

2.1 Učení s učitelem

Během tohoto způsobu učení jsou jako trénovací data předloženy jak vstupy, tak i správné výstupy. Učení probíhá v iteracích. Intervalu, ve kterém dojde k přečtení všech trénovacích dat aspoň jednou, se říká epocha. Jednotlivých epoch může být zapotřebí až tisíce, což závisí především na komplexnosti problému. Při učení s učitelem program ví, jakých výsledků má dosáhnout, protože je má od začátku předložené. Na základě porovnání těchto výsledků a jejich odchylky si upravuje váhy jednotlivých neuronů.[1]

2.2 Učení bez učitele

Při učení bez učitele dostane program pouze vstupní data. Jelikož nemá žádná data, podle kterých by si ověřil správnost svého výsledku, tak se výsledky snaží shlukovat na základě podobnosti jednotlivých elementů. Tento způsob se využívá méně než jeho protichůdce, ale našel si oblibu v kybernetické bezpečnosti.[1]

2.3 Využití machine learningu

Strojové učení lze, jak již bylo zmíněno, využít prakticky kdekoliv. Když se podíváme spíše na technicky zaměřené pojmy, než na věci z běžného života, tak se dá použít například u shlukování, regrese či klasifikace.

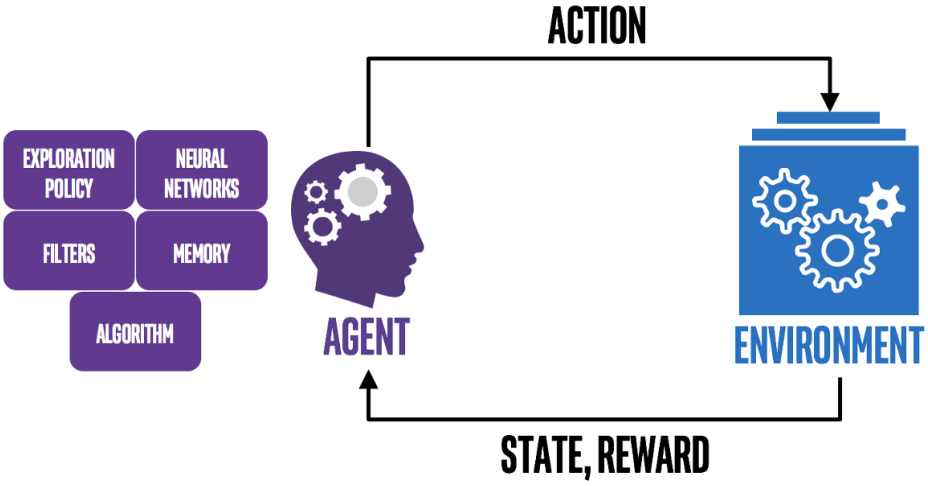
- Klasifikace - u klasifikace se strojové učení používá například k rozpoznání různých věcí v prostředí. Třeba zda se na fotce vyskytuje kočka, pes nebo třeba kůň. Klasifikace využívá učení s učitelem.
- Shlukování - shlukování, jak bylo zmíněno výše, je typický příklad učení bez učitele. Program obdrží data, která se následně snaží rozdělit do skupin na základě jejich podobnosti. Tomuto jevu se říká shlukování a typickým příkladem může být rozdělení nad známými Iris daty.[2]
- Regrese - Regrese je příkladem učení s učitelem. Je to statistická metoda, která modeluje vztah mezi závislými a nezávislými daty, jako například teplota, věk, výplata. Pomocí strojového učení dokážeme předpovědět, jak se tato data budou dále vyvíjet na základě dat z předchozích období.[3]

Kapitola 3

Reinforcement Learning

Zpětnovazební učení, známo pod svým anglickým názvem Reinforcement Learning, je způsob strojového učení, u kterého nejsou potřebná trénovací data. Toto je obrovskou výhodou, protože vytvoření trénovacích dat může být někdy velmi obtížné.[4] Zpětnovazební učení má 4 hlavní znaky:

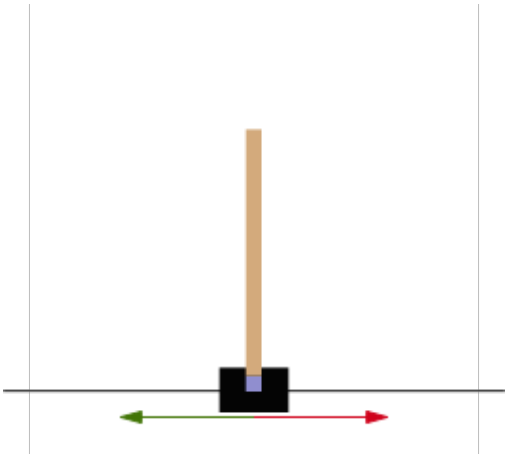
- Agent - Program, který se trénuje aby dělal svou práci správně.
- Prostředí - Simulace reálného nebo virtuální světa, ve kterém agent vykonává akce.
- Akce - Jeden tah, který agent vyvolá v prostředí a tím změní jeho stav.
- Odměna - Ohodnocení akce, které říká, zda měla pozitivní nebo negativní dopad na prostředí.



Obrázek 3.1: Model RL [4]

Zpětnovazební učení je, jak již vyplývá z názvu, založeno na zpětné vazbě neboli odměnách, kterými odměňujeme agenta. Odměna může být kladná, nebo záporná. Při procesu učení agent vykonává akce, které jsou dle něho nejlepší. Po vykonání je tato akce vyhodnocena a agentovi je vrácena odměna. Na základě této odměny agent přehodnotí své budoucí rozhodnutí. Mějme například agenta, který se vyskytuje v nějakém námi vybraném prostředí. V tomto prostředí může agent vykonávat akce a tím ho ovlivnit. Agent v každé iteraci T pozoruje prostředí St. Na základě stavu tohoto prostředí provede akci At a tím převede prostředí do nového stavu $St + 1$. Za provedení této akce je mu navrácena odměna Rt.[5]

Jedním z takových příkladu může být hra Cart-Pole. [6] Prostředí této hry nabízí knihovna OpenAIGym. V tomto prostředí má agent možnost provést dvě akce. Pohnout vozíkem doleva nebo doprava. Po provedení této akce se také pohne tyč, která stojí na vozíku. Úkolem programu je hýbat s vozíkem tak, aby se tyč z vozíku nepřevrhla.



Obrázek 3.2: Vyobrazení cart-pole problému [6]

3.1 Markovův rozhodovací proces

Markovův rozhodovací proces je matematický model pro rozhodovací problémy, ve kterých jsou rozhodnutí částečné ovlivnitelná a náhodná. Tento model se často využívá pří zpětnovazebním učení.

Předtím, než bude možné podrobněji vysvětlit co je Markovův rozhodovací proces, je potřeba znát některé výrazy. Konkrétně je potřeba vědět, co je Agent, Prostředí, Akce, Odměna a Policy. První 4 výrazy byly popsány v předchozí kapitole. Policy by se dala popsat jako chování během procesu výběru správné akce v určitém stavu. Můžeme tedy říct, že se jedná o mapování stavů na akce, které mají být v tomto určité stavu uskutečněny.[5]

3.2 Markovova properta

Stav se dá brát jako Markovova properta pouze, když platí následující pravidlo:

$$P[S_t + 1, |S_t] = P[S_t + 1|S_1, S_2, ...S_t] \tag{3.1}$$

Markovova properta říká, že momentální stav závisí pouze na jednom předchozím stavu a nezávisí na stavech, které se udály předtím. Například si představte, že agent stojí v místnosti. Poté se posadí na židli a vezme do ruky tužku. Agent je ve stavu, ve kterém sedí na židli a drží tužku. Tento stav je závislý pouze na stavu, kdy agent pouze seděl na židli. Stav kdy agent stál, není v tomto případě vůbec podstatný.[7]

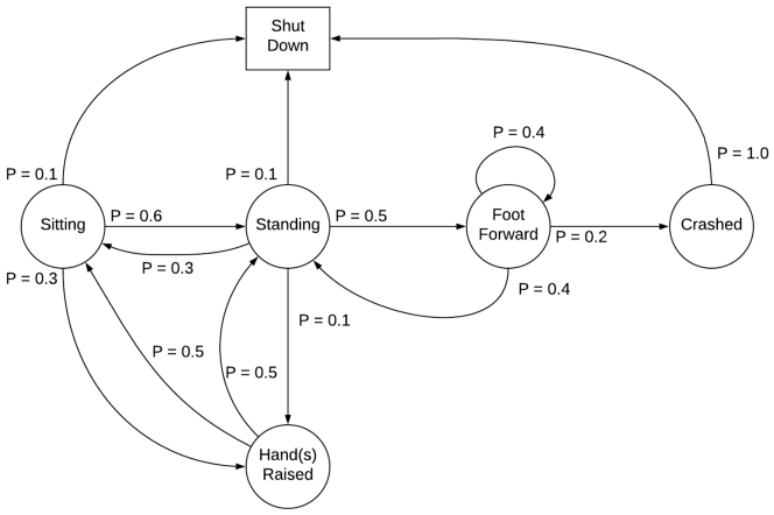
3.3 Markovův proces a markovův chain

$$P_{ss'} = P[S_t + 1 = S'|S_t = S] \tag{3.2}$$

Markovův proces je definovaný jako (S,P) kde:

- S - konečná množina stavů
- P - množina pravděpodobností přechodů do dalších stavů

Je to posloupnost stavů *S1,S2,S3..*, ve které musí všechny stavy být Markovova properta. Pravděpodobnost přechodu do dalšího stavu *Pss'* je pravděpodobnost přechodu ze stavu *s* do stavu *s'*. [7]



Obrázek 3.3: Markovův chain [7]

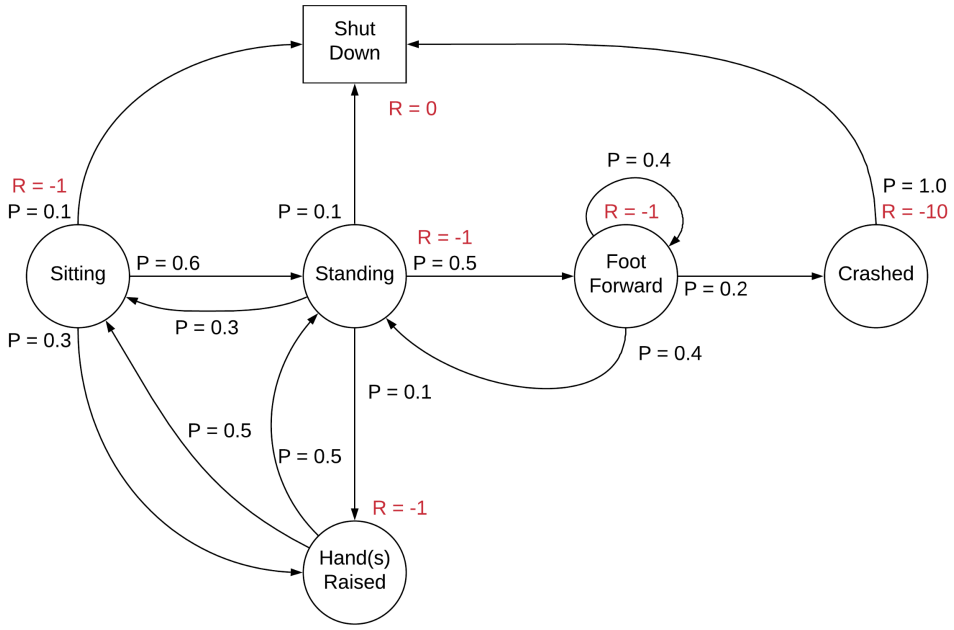
3.4 Markovův reward proces

$$R_s = E[t+1|S_t = S] \tag{3.3}$$

Markovův reward proces je Markovův proces (dále jen MRP) s ohodnocením, které říká, jak moc odměn tento proces získal. MRP je definovaný jako (S,P,R) [7], kde:

- S - konečná množina stavů
- P - množina pravděpodobností přechodů do dalších stavů
- R - funkce odměn

Funkce odměn udává, jak velkou okamžitou odměnu agent dostane v právě se nacházejícím stavu.[7]



Obrázek 3.4: Markova odměna [7]

Všechno dohromady dává Markovův rozhodovací proces (dále jen MDP), což je MRP s rozhodnutími. MDP je definován jako (S,A,P,R) [7], kde:

- S - konečná množina stavů
- A - množina akcí
- P - množina pravděpodobností přechodů do dalších stavů
- R - funkce odměn

Skoro všechny rozhodovací problémy můžeme řešit pomocí MDP, je pouze potřeba identifikovat množinu stavů a funkcí.

3.5 Výzvy v reinforcement learningu

Jedním z nejsložitějších problémů je připravení simulovaného prostředí, ve kterém se bude agent učit. Porovnejme mezi sebou například prostředí pro hraní šach a prostředí pro autonomní řízení auta. Pokud se jedná o šachy, je potřeba vytvořit herní desku a stanovit pravidla hry jako takové. Pokud bychom chtěli naučit auto nechat autonomně řídit v reálném světě, musíme vytvořit simulaci právě tohoto reálného světa, ve kterém se auto bude učit. Vytvoření šachové desky tedy můžeme označit jako velmi jednoduchý úkol v porovnáním s vytvořením simulace pro autonomní řízení.[8]

Kapitola 4

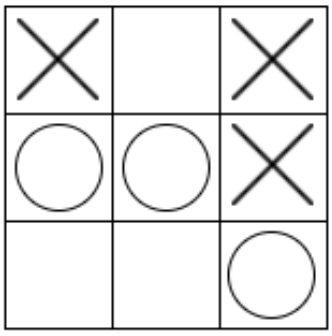
Minimax Algoritmus

První algoritmus, který jsem se rozhodl implementovat se nazývá minimax. Nejedná se sice o algoritmus, který využívá strojového učení, ale na začátek, pro pochopení, jak vůbec funguje rozhodování umělé inteligence, mi přišel jako vhodný adept. Minimax algoritmus funguje na bázi rozhodování takovým způsobem, že nasimuluje všechny možné tahy ze stavu, ve kterém se právě nachází a vyhodnotí, který z nich je nejlepší. Vyhodnocování funguje ve dvou fázích, které se navzájem střídají.

1. MAX -> Tah hráče - algoritmus vybere nejlepší možnou variantu pro nás, takže maximalizuje naše šance na výhru.

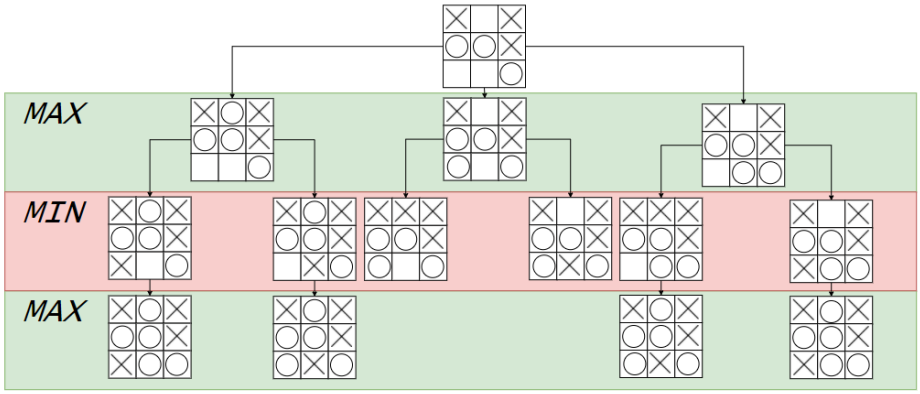
2. MIN -> Tah soupeře - algoritmus vybere nejlepší možnou variantu soupeře, takže nejhorší možnou variantu pro nás a minimalizuje tak naše šance na výhru.

Vyhodnocování vždy začíná naším tahem. Představte si například, že se hra nachází právě v tomto stavu a my chceme vědět jaký tah je nejlepší.



Obrázek 4.1: Aktuální stav herní plochy

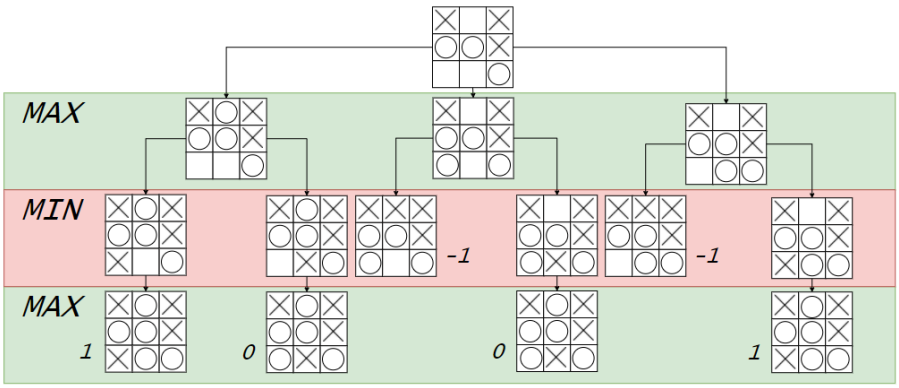
Na řadě je hráč se znakem kolečka. Algoritmus tedy nasimuluje všechny možné tahy, které může zahrát. Poté z každého vzniklého stavu nasimuluje všechny tahy nepřítele. Takto algoritmus pokračuje a vzniká struktura podobná stromu. Algoritmus končí, pokud je ve všech větvích hra ukončena. Po všech simulacích může strom stavů vypadat takto.



Obrázek 4.2: Nasimulované všechny možné tahy

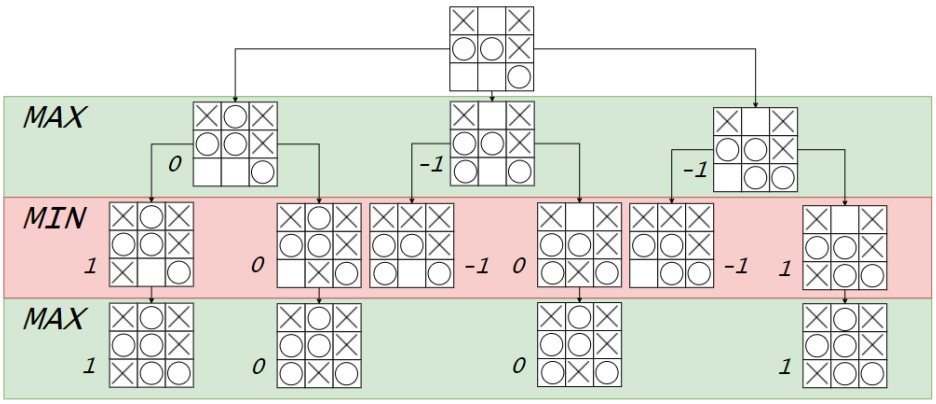
Jakmile je simulace ukončena, algoritmus ohodnotí každý finální stav z pohledu hráče. Ohodnocení může vypadat následovně:

1. Výhra 1
2. Prohra -1
3. Remíza 0



Obrázek 4.3: Ohodnocení všech finálních tahů

Po vyhodnocení všech stavů začne algoritmus propagovat výsledky do vyšších vrstev. Z MAX vrstvy se vždy vybírá nejvyšší nejlepší hodnota pro nás. Z MIN vrstvy se vždy vybírá nejnižší nejhorší hodnota pro nás. Jakmile jsou výsledky propagovány až do první vrstvy, vybere algoritmus stav s nejvyšší hodnotou a tento tah vyhodnotí jako nejlepší.



Obrázek 4.4: Hodnoty po propagaci zpět do první vrstvy

V našem případě vybere hned první stav a nejlepším tahem je pro tuto situaci zahrání kolečka na pozici [1,2]. Pokud se stane, že se vyhodnotí více tahů jako nejlepší volba, může se algoritmus chovat dvěma způsoby.

- Deterministicky - Algoritmus vždy vybere stejnou variantu, tu kterou vyhodnotil jako první.
- Nedeterministicky - Algoritmus vždy vybere jinou variantu a to takovým způsobem, že ze všech označených jako nejlepší, náhodně vybere jednu.

I když je algoritmus velmi účinný, protože vždy vybere ten nejlepší možný tah, má jeden velký problém. S většími a složitějšími hrami exponenciálně narůstá počet možných stavů, které musí nasimulovat, takže se algoritmus stává hodně pomalým. I přes to se však na algoritmus dají použít různá vylepšení, která mohou problémy alespoň částečně odstranit.

4.1 Transposition table

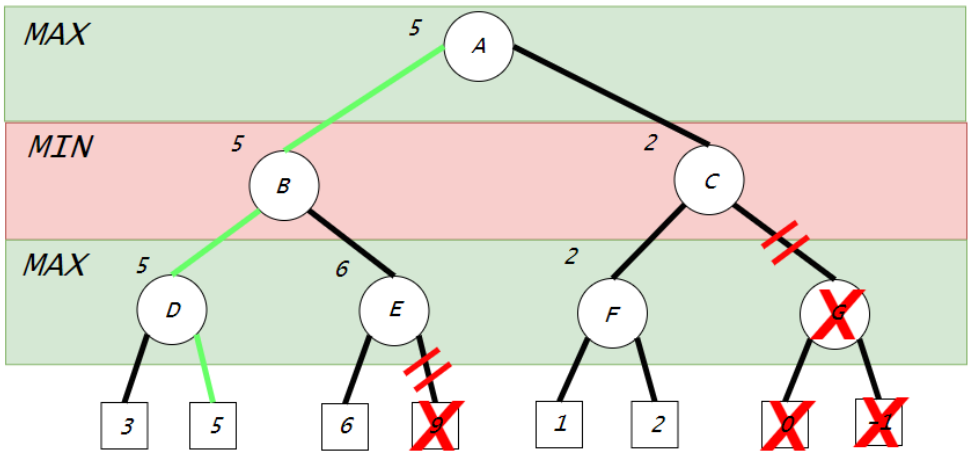
Jedním z těchto vylepšení může být takzvaná transpoziční tabulka.[9] Tato tabulka uchovává veškeré stavy, kterými už algoritmus prošel a ukládá si k nim nejlepší možný tah. Pokud se hra ocitne ve stavu, ve kterém už v minulosti byla, tak algoritmus se místo simulace podívá do tabulky a vybere z ní již dříve zjištěný nejlepší tah. Jedním z efektivních způsobů je ukládat data jako dvojice [stav, nejlepší tah]. Abychom nemuseli ukládat celý stav herní plochy, tak využijeme hašování a herní stav, který bude pravděpodobně nějaké pole hodnot zahašujeme nějakou funkcí.

4.2 Alpha beta pruning

Dalším způsobem, jak urychlit minimax algoritmus je alfa-beta ořezávání.[10] V tomto případě nám přibydou dvě proměnné.

- Alfa - Nejlepší MAX hodnota pro prozkoumané možnosti.
- Beta - Nejlepší MIN hodnota pro prozkoumané možnosti.

Tato optimalizační metoda funguje tak, že při simulování a prohledávání porovná výslednou hodnotu s hodnotou z předchozího tahu. Pokud algoritmus zjistí, že je hodnota horší, než u již dříve zjištěných možností, nebude tuto větev dále prohledávat a odřízne ji.



Obrázek 4.5: Ukázka výsledku s využitím alfa-beta ořezáváním

Kapitola 5

Q-Learning

První algoritmus, který se pravděpodobně každému vybaví při zmínce o zpětnovazebním učení, je Q-Learning, celým názvem Quality Learning. Q-Learning často bývá prvním algoritmem, protože je lehký jak k pochopení, tak k implementaci.

Q-Learning je off-policy algoritmus zpětnovazebního učení, který hledá nejlepší akci v určitém stavu. Algoritmus se bere jako off-policy, protože se agent učí z akcí, které nastaly v jiném stavu, než ve kterém se právě nachází. Jinak řečeno Q-Learning se snaží maximalizovat celkovou odměnu.[11]

Není to náhoda, pokud se vám tento postup zdá povědomý. Q-Learning totiž jako zpětnovazební algoritmus využívá MDP. Jako příklad můžeme použít hru TicTacToe, která byla použita i pro příklad u minimaxu. Konečnou množinou stavů S jsou všechny legální stavy, které mohou ve hře nastat. Množinou akcí A jsou tahy, které může hráč z aktuálního stavu provést. Množina pravděpodobností přechodů P je množina, podle které se bude Q-Learning rozhodovat, jakou akci si vybere. A nakonec funkce odměny R, podle které se bude upravovat množina pravděpodobností na základě předešlých tahů.

Q-Learning spočívá v tom, že agent pracuje s dvourozměrnou maticí o velikosti stavy x akce zvanou Q-table. Tato matice bude sloužit jako reference k vybrání nejlepší akce. Na začátku jsou všechny hodnoty této matice inicializovány na nějakou hodnotu. Tato tabulka je po každé akci aktualizována.

		Akce		
	A1	A2	A3	A4
S1	0.3	0.5	0.1	0.1
S2	0.5	0.2	0.3	0
S3	0.1	0.8	0	0.1
S4	0.05	0.5	0.35	0.55
...	...	...	...	...
Sn	0	0	0	0

Stavy

Obrázek 5.1: Q Tabulka

Po vytvoření tabulky algoritmus běží v jednotlivých iteracích neboli epizodách. V jedné epizodě agent provede x počet kroků, ve kterém každý krok obsahuje tuto posloupnost akcí:

- Agent zjistí stav, provede akci a získá odměnu
- Agent vybere akci pomocí Q-table nebo náhodně
- Aktualizace tabulky

5.1 Provedení a vykonání akce

Při vybrání akce se agent může rozhodnout dvěma způsoby. Prvním způsobem je použít Q-table jako referenci a podívat se na všechny možné akce z aktuálního stavu. Agent poté vybere akci s největší hodnotou. Tomuto způsobu se říká exploitace.[11]

Druhým způsobem je náhodný výběr. Místo výběru nejlepší možné akce, agent vybere jakoukoliv možnou akci bez toho, aby se díval na její hodnotu. Tento způsob se nazývá explorace. I když to tak nevypadá, vybrat akci pomocí explorace je velmi důležité pro to, aby agent objevil nové stavy, ke kterým by se pomocí exploitace dříve nedostal. Explorace a exploitace by měly být v rovnováze. K tomu využijeme parametr ϵ, který nastavíme podle toho, jak moc chceme, aby agent exploroval či exploitoval. V nejlepším možném případě je epsilon nastaven tak, aby agent ze začátku spíše exploroval a později, když už prozkoumal dostatek stavů, začal spíše exploitovat.[11]

5.2 Aktualizace Q-table

Jak již bylo zmíněno dříve, aktualizace probíhá při každém provedení akce a končí na konci epizody. Konec epizody může znamenat dosažení nějakého bodu, ukončení partie deskové hry nebo splnění nějakého cíle. Agent se moc nenaučí po první epizodě, ale po x epizodách a aktualizacích začne Q-table nabírat optimálních hodnot.[11] Aktualizace probíhá pomoci tohoto vzorečku:

$$Q[state, action] = Q[state, action] + lr * (reward + \gamma * np.max(Q[new_s state, :]) - Q[state, action]) \tag{5.1}$$

Ve vzorečku se pracuje s jistými neznámými hodnotami, které jsou popsány níže. Aktualizace funguje tak, že upravujeme hodnotu v tabulce na základě rozdílu mezi spočítanou novou a starou hodnotou. Novou hodnotu počítáme pomocí proměnných gamma a odměny a upravujeme pomocí proměnné, kterou nazýváme učící faktor.

- Učící faktor - Learning rate, občas nazývaný také jako alfa, definuje jak často chceme akceptovat novou hodnotu oproti staré. Ve vzorečku pracujeme s rozdílem těchto dvou hodnot a násobíme ho touto proměnou. Tato nová hodnota je poté přičtena k té předešlé.[11]
- Gamma - Gamma je faktor, který slouží k vyrovnání okamžité a budoucí odměny. Gammu aplikujeme na budoucí odměnu. Většinou nabývá hodnot mezi 0.8 - 0.99.[11]
- Odměna - Hodnota, kterou agent dostane za vykonání určité akce v určitém stavu. Odměna může být získána při každém provedení akce, nebo až při poslední akci.[11]

Kapitola 6

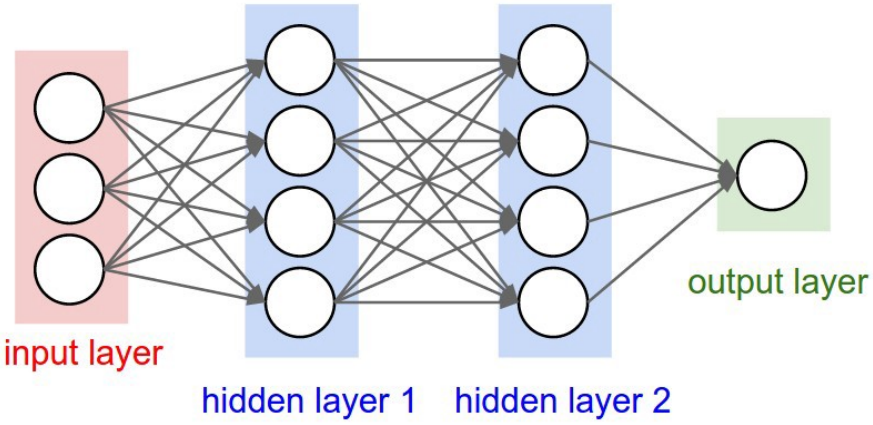
Deep Q-Learning

Q-Learning se může zdát jako dokonalý adept pro zpětnovazební učení, ale má také svá omezení. Jedno z těchto omezení je velikost tabulky, kterou tento algoritmus používá pro zjištění dalšího tahu. Když například vezmeme v potaz TicTacToe o klasické velikosti 3x3, tak lehkým výpočtem 3^9 zjistíme, že existuje 19,683 možných herních stavů. [12] To ovšem zahrnuje také herní stavy, které nejsou legální neboli dosažitelné. Jedná se například o stav, ve kterém mají oba hráči 3 políčka v řadě. I kdybychom počítali s těmito ilegálními stavy, tak dnešní výpočetní technika by neměla mít nejmenší problém pracovat s tabulkou o takové velikosti. Problém nastáva v případě komplexnějších her, jako například u deskové hry GO, která má počet legálních herních stavů přibližně $2,1{*}10^{170}$[13]. Pro představu, počet známých atomů se odhaduje na $1 * 10^{80}$, takže s takovou tabulkou už by bylo prakticky nemožné pracovat. Z tohoto důvodu připadá v uváhu použití neuronových sítí, které místo toho, aby si udržovaly informace o všech možných stavech a na základě toho vybraly nejlepší tah, dokážou tento tah předpovědět.

6.1 Neuronové Sítě

Neuronové sítě jsou ve své podstatě algoritmem, který se dá přirovnat k chování našeho mozku. Když člověk něco vidí nebo slyší, tak to jsou prakticky data, která musí nějak zpracovat. Tato data se zpracují v mozku za pomoci neuronů a následně člověk rozpozná, co konkrétně vidí nebo slyší. A přesně takovým způsobem fungují neuronové sítě. Samozřejmě ne doslova, protože vytváříme umělé neurony, které se pouze snaží napodobit chování pravých neuronů.[14][15]

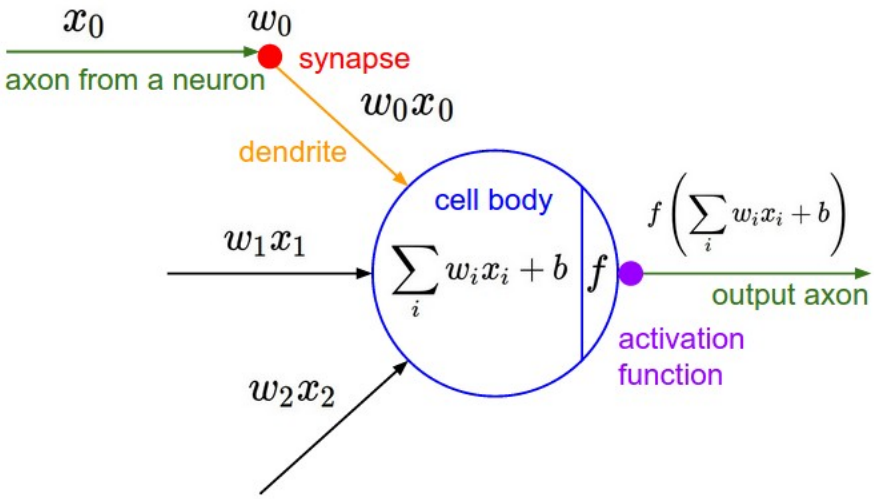
Neuronová síť se skládá ze tří základních vrstev. První je vstupní vrstva, která přijímá data pro zpracování. Druhá vrstva je skrytá a probíhá v ní zpracování vstupních dat. Takových skrytých vrstev může být za sebou několik. Poslední vrstva je vrstva výstupní, na které se nám zobrazí výsledek. Výsledky mohou být ze začátku velmi nepřesné, protože stejně jako u člověka musí nejdříve proběhnout nějaká fáze učení.



Obrázek 6.1: Neuronová Síť [16]

Každá tato vrstva se skládá z několika neuronů. Tomuto neuronu se také říká node nebo unit.[15] Každý tento neuron má jeden nebo více vstupů, a právě jeden výstup. Jako vstup mohou sloužit vstupní data nebo výstupy jiných neuronů. Kromě vstupu a výstupu obsahuje také dva parametry, které slouží k učení:

- Weight - Pro každý vstup existuje jedna tato hodnota. Určuje jak důležitá bude vstupní hodnota pro výpočet neboli jak moc bude ovlivňovat výstup neuronu.[17].
- Bias - Bias by se dal považovat za další vstupní parametr. Je to konstanta, pomocí které dokážeme ovlivnit konečný výsledek neuronu. Jelikož se jedná o vstup má také přiřazenou svoji váhu.[17].



Obrázek 6.2: Neuron [16]

6.1.1 Aktivační funkce

V každém neuronu probíhá výpočet jako suma všech vstupů vynásobených svou váhou, ke kterému na konec přičteme bias. Na tuto výslednou hodnotu je následně použita aktivační funkce, jejíž výsledkem je výstup neuronu.[15]

$$f(\sum_i w_i * x_i + b) \tag{6.1}$$

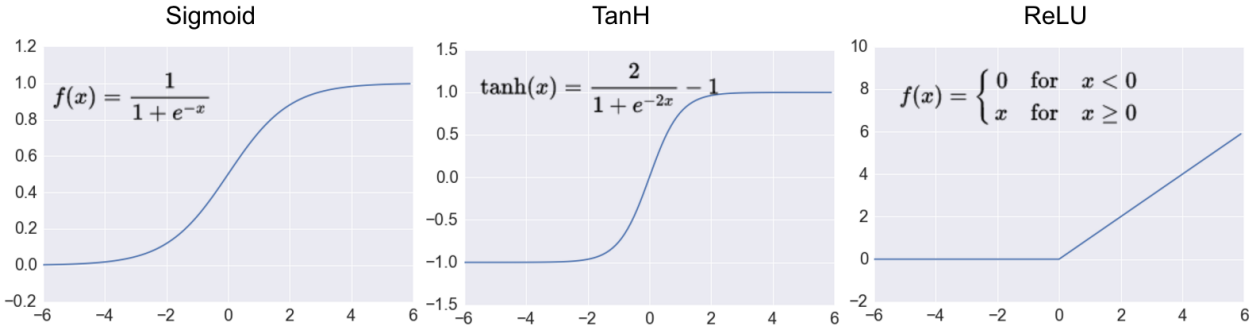
Aktivační funkce je nelineární. Je to z toho důvodu, že skoro všechna reálna data nejsou lineární a my chceme, aby se neuronová síť dokázala naučit s těmito nelineárními daty pracovat. Každá aktivační funkce bere na vstupu jednu proměnnou a provede nad ní fixní operaci. Existuje několik aktivačních funkcí. Mezi ty nejpoužívanější patří:

- Sigmoid - Funkce sigmoid vezme reálné číslo na vstupu a zařadí jej do intervalu (0,1) [15].
- Softmax - Funkce softmax se nejvíce používá u klasifikace. Používá se na výstupní vrstvě kde zajistí, aby se výsledné pravděpodobnosti rovnaly jedné.[15]

$$P(true) + P(false) = 1 \tag{6.2}$$

- TanH - Funkce TanH funguje stejně jako funkce sigmoid, akorát zařadí výsledek do intervalu (-1,1) [15].
- ReLU - Funkce ReLU vezme reálné číslo na vstupu a pokud je negativní nahradí ho 0. Pokud není, nedělá s ním nic.[15]

$$f(x) = max(0, x) \tag{6.3}$$



Obrázek 6.3: Ukázka aktivačních funkcí [18]

6.1.2 Průběh neuronové sítě

Celý chod neuronových sítí má dvě hlavní fáze, které se navzájem střídají. První z nich se nazývá feed forward propagation neboli forward propagation a druhá backward propagation neboli back propagation.

Na začátku jsou všechny váhy v každém neuronu zvoleny náhodně. To má za příčinu, že první výsledky neuronové sítě nejspíše nebudou moc přesné. Berme v potaz, že když na vstupu máme například hodnoty 20, 35 a 80, tak náš požadovaný výstup je [1,0]. Neuronová síť poté provede na vstupních neuronech tento výpočet:

$$Y = f(20 * w1 + 35 * w2 + 80 * w3 + bias) \tag{6.4}$$

Tyto výsledky se protlačí dále do dalších vrstev, dokud nedostaneme konečný výsledek ve vrstvě výstupní. Tato fáze se nazývá forward propagation. Například při původním náhodném nastavení vah dostaneme na výstupu hodnotu [0.6,0.4]. Tento výsledek je dost daleko od našeho požadovaného. Neuronová síť poté spočítá, jak velkou udělala chybu. V našem případě je chyba 0.4 a -0.4. Tuto chybu pošle opačným směrem zpět do neuronové sítě.

Tato druhá fáze se nazývá back propagation a během ní dochází k úpravě vah. Tyto hodnoty ohledně chyby budou proplouvat až na začátek neuronové sítě a v každém neuronu se podle nich budou upravovat váhy tak, aby při dalším výpočtu nedošlo k tak velké chybě. Těchto výpočtu bude samozřejmě potřeba obrovské množství s různými parametry, než si síť dokáže správně nastavit tyto váhy.

6.2 DQN

Jak již bylo řečeno, tak Q-Learning není vhodný, pokud máme miliony různých stavů a tisíce možných akcí, kvůli velikosti tabulky, kterou by si musel udržovat. Deep Q-Network je algoritmus, který se tomuto problému snaží předejít pomocí kombinace právě zmíněného Q-Learningu a neuronových sítí.[19]

Neuronové sítě jsou využity k tomu, aby aproximovaly Q-funkci pro vypočtení nejlepší akce a z tohoto důvodu není nutné udržovat Q-table. Reálně jsou v tomto alqoritmu využity 2 neuronové sítě. První, která se označuje jako hlavní, reprezentuje váhu vektoru θ' a používá se pro odhadnutí Q-hodnot pro aktuální stav a akci. Druhá síť se nazývá cílová síť, která je parametrizovaná váhou vektoru θ' a má přesně stejnou architekturu jako síť hlavní. Tato síť se používá k vypočtení Q-hodnot ne aktuálního stavu, ale budoucího stavu a akce. Tato síť není aktualizována jako hlavní síť, ale místo toho jsou většinou každých 10000 iterací váhy z hlavní sítě okopírovány. Jinak řečeno hlavní síť převede své naučené zkušenosti na cílovou síť, která je poté využije pro lepší odhady.[19]

Dále ještě síť potřebuje Bellmanovu rovnici a funkci pro výpočet ztráty. V případě DQN se ztráta spočítá jako druhá mocnina rozdílu obou stran Bellmanovi rovnice.[19]

$$Q(s, a, \theta) = r + \gamma MaxQ_{a'}(s', a', \theta') \tag{6.5}$$

$$L(\theta) = E[(r + \gamma MaxQ_{a'}(s', a', \theta') - Q(s, a, \theta))^2] \tag{6.6}$$

Pro zlepšení učení DQN lze využít takzvaný experience replay.[20] Tato technika funguje na bázi toho, že se síť neučí pouze ze zkušeností z poslední iterace, ale ze všech. Využívá se k tomu replay buffer, do kterého ukládáme jednotlivé kroky ve formátu (S, A, R, S') [19] kde:

- S - Počáteční stav.
- A - Vybraná akce v tomto počátečním stavu.
- R - Odměna, kterou agent dostal za vykonání akce.
- S' - Stav do kterého se po provedení akce agent dostal.

Až bude v bufferu dostatek záznamu, bude z něho v pravidelných intervalech vybíráno určité množství, které si agent přehraje a bude se z nich učit. Záznamy jsou vybírány náhodně tak, aby se agent mohl učit také z akcí, které se například nestanou tak často.

6.3 Double DQN

DQN není sama o sobě dokonalá, takže existují rozšíření, která tento algoritmus v některých ohledech zdokonalují. Jedním z těchto rozšíření je Double DQN.[21] Jeden z problému totiž je, že DQN přehlíží reálnou odměnu. Q-hodnota si poté myslí, že dostane větší odměnu, než dostane ve skutečnosti. Double DQN toto řeší jednoduchým trikem, a to malou změnou Bellmanovi rovnice. To má za příčinu, že nejdříve, stejně jako u DQN, hlavní sít zjistí, která je nejlepší akce z aktuálního stavu, ale poté cílová síť tuto akci ohodnotí tak, aby zjistila její reálnou hodnotu. Díky této změně by síť měla dosahovat lepších výsledků.

$$Q(s, a, \theta) = r + \gamma Q(s, agrmax_{a'}(Q(s', a', \theta), \theta') \tag{6.7}$$

6.4 Dueling DQN

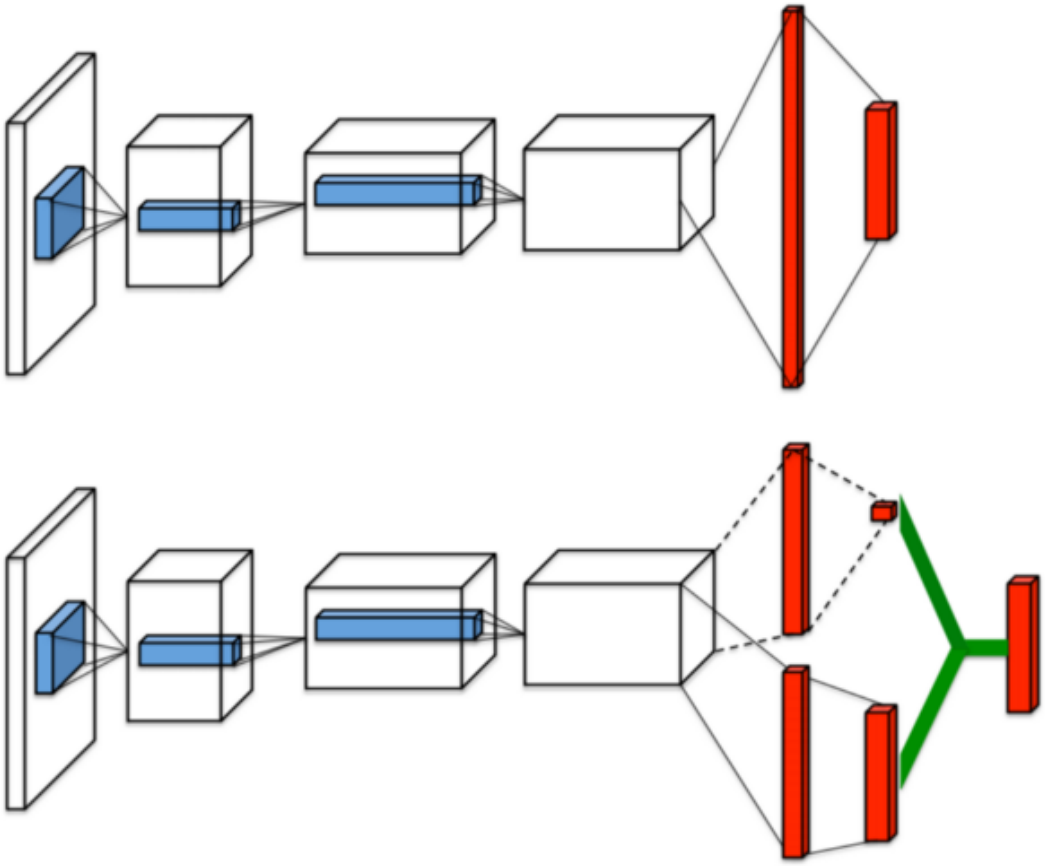
Druhé rozšíření DQN je Dueling DQN.[21] Toto rozšíření rozděluje Q-hodnoty na dvě různé části, funkci hodnoty a výhody. Každá tato funkce má svou podstatu. Funkce hodnoty zjistí, jak velkou odměnu agent po vykonání akce získá z aktuálního stavu. Funkce výhody poté spočítá, o kolik je tato akce lepší než všechny ostatní. Kombinací těchto dvou funkcí dostaneme výslednou Q-hodnotu.

$$Q(s,a) = V(s) + A(s,a) \tag{6.8}$$

K čemu je toto rozdělení dobré? Berme v úvahu, že agent je ve stavu S a má na výběr z akcí A1, která ho posune do stavu S1 a A2, která ho posune do stavu S2. A1 má větší hodnotu než A2, takže podle DQN je správné vybrat A1. Jenže může nastat situace, ve které jsou lepší možnosti dalšího postupu z A2, než z A1, takže je v reálné situaci výhodnější zvolit právě akci A2. Přesně proto je zde funkce výhody, která zjišťuje jestli je akce opravdu nejlepší.

Menším problémem této sítě je fakt, že ji nedokážeme trénovat jednoduchou sumou hodnoty a výhody. Pokud máme Q = V+A tak je nemožné zjistit hodnoty V a A ze známé hodnoty Q. Například když Q bude rovno 65 tak dostaneme rovnici 65 = V + A. V tomto případě je nemožné zjistit jaké dvě hodnoty byly použity k dosažení našeho výsledku, protože zde existuje nekonečně mnoho řešení. K vyřešení toho problému byl vymyšlen trik. Můžeme předpokládat, že nejvyšší Q-hodnota je rovna hodnotě V, čímž uděláme nejvyšší hodnotou ve funkci výhody 0 a všechny ostatní menší než 0. Toto zapříčiní, že budeme vědět jakou hodnotu má V a budeme schopni dopočítat zbytek.[21] Trénovací funkce může tedy vypadat takto:

$$Q(s,a) = V(s) + (A(s,a) - max_{a' \epsilon |A|} A(s,a)) \tag{6.9}$$

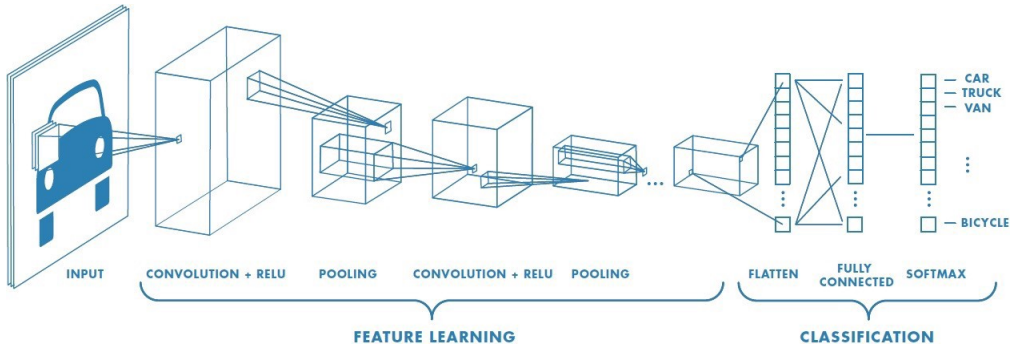


Obrázek 6.4: DQN vs Dueling DQN [21]

Kapitola 7

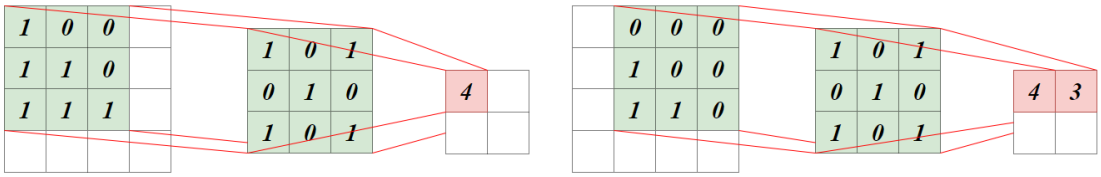
Konvoluční sítě

Konvoluční sítě jsou typ neuronových sítí, které požadují na vstupu fotku nebo nějaký obrázek.[22] I když se tento druh sítí primárně používá pro klasifikaci, například rozpoznání jestli je na obrázku pes nebo kočka, lze je také využít při spojení s hlubokými sítěmi. Středem konvolučních sítí je konvoluční vrstva, podle které jsou sítě pojmenovány. Tato vrstva provádí operaci, která může být odhadnutá z názvu, zvanou konvoluce. Celý proces konvolučních sítí funguje tak, že vstupní data projdou jednou či více konvolučními vrstvami. Po každé konvoluční vrstvě se nachází takzvaná pooling vrstva. Na konec se data sjednotí do jednodimenzionálního pole, se kterým se dále pracuje stejně jako tomu bylo u DQN.



Obrázek 7.1: Schéma konvoluční sítě [22]

Jak bylo řečeno, konvoluční sítě berou jako vstup nějaký obraz. Na tento obraz poté používají různé filtry/kernely, které obraz rozdělují na jednotlivé části. V těchto částech hledají jednotlivé rysy či vlastnosti, podle kterých dokážou rozpoznat o co se vlastně jedná. Tyto jednotlivé části vytvoří tak zvanou mapu rysů, která zvýrazňuje jak moc se v jednotlivých částech rysy nachází.[22][23] Nejlépe to půjde popsat pomocí obrázku:



Obrázek 7.2: Zpracování dat v konvolučních sítích

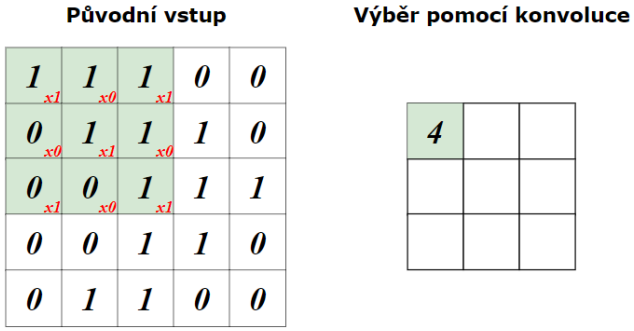
V našem případě můžeme konvoluční sítě použít na rozpoznání stavu herního pole a podle toho vyhodnotit, jaká je nejlepší možná akce. Způsob učení zůstane prakticky stejný jako tomu bylo u DQN. Co se změní jsou vstupní data a zpracování těchto vstupních dat.

7.1 Convolutional layer

Při vytváření konvoluční vrstvy je třeba specifikovat, jak vypadají vstupní data. Vstupní data by měla být v tomto formátu (n, x, y, channels) kde:

- N - počet vstupních dat
- X - šířka vstupních dat (velikost obrázku)
- Y - výška vstupních dat (velikost obrázku)
- Channels - počet barevných kanálu vstupních dat

Dále se musí definovat velikost filtru neboli po jakých částech bude konvoluce probíhat. Například naše vstupní data mají velikost 5x5 a vybereme velikost filtru 3x3.[23] Poté bude zpracování probíhat tak, jak je vidět na následujícím obrázku:



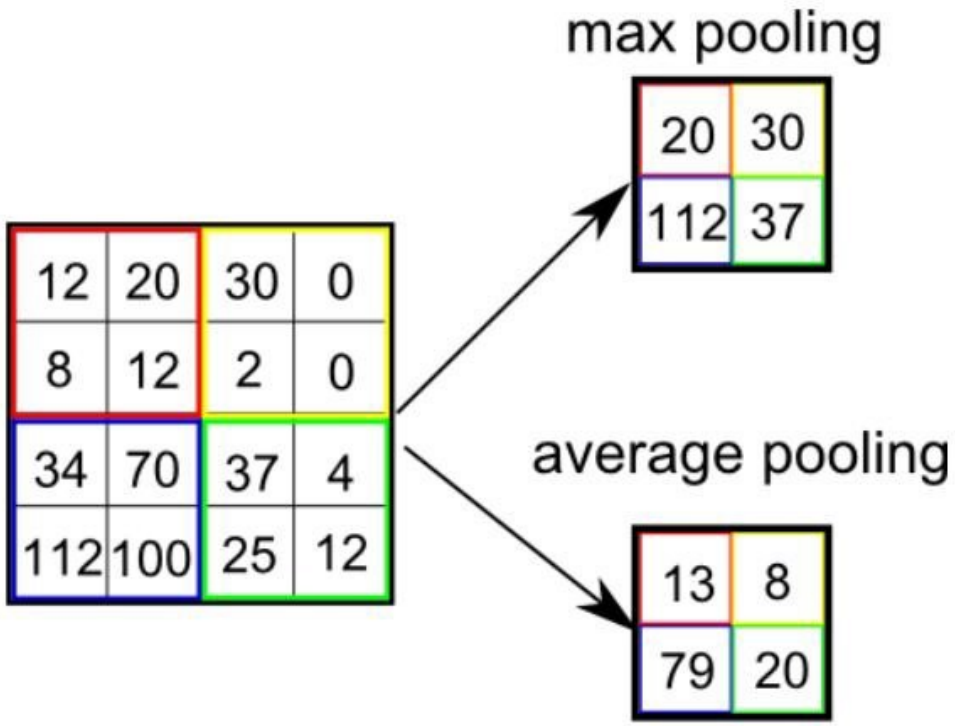
Obrázek 7.3: Zpracování dat v konvolučních sítích

Tento výpočet by při větším počtu barevných kanálů probíhal pro každý kanál zvlášť. Úkolem této vrstvy je vlastně zjednodušení vstupních dat do formátu, který je snažší pro další zpracování, bez toho, aby se ztratila informace o vlastnostech, podle kterých bude síť na konci rozhodovat.

7.2 Pooling layer

Podobně jako konvoluční vrstva, tak i pooling vrstva má na starost redukovat velikost zpracovávaných dat a tím pádem urychlit výpočet. Tato vrstva funguje tak, že z jednotlivých částí mapy rysů vybere pouze určité rysy, vlastnosti.[23] Výběr rysů lze provést dvěma způsoby.

- Max pooling - Z jednotlivých částí vybírá maximum neboli rys, který je vidět nejvíce. [23]
- Average pooling - Z jednotlivých částí vybírá průměr všech hodnot z části, nad kterou pracuje. [23]



Obrázek 7.4: Způsoby vybrání rysů [23]

Je viditelné, že max pooling funguje pro rozpoznání obrazu o dost lépe než average pooling. Po tomto zpracování můžeme data překonvertovat do jednodimenzionálního pole a poslat je jako vstupní data do normální neuronové sítě.

Kapitola 8

Implementace

Na začátku této kapitoly budou popsány technologie, které jsem využil k vypracování této diplomové práce. Poté popíšu architekturu projektu a podrobně rozeberu jednotlivé části. Na konec popíšu implementace jednotlivých algoritmů, u kterých se primárně zaměřím na problémy, se kterými jsem se potýkal.

8.1 Využité technologie

8.1.1 Python

K implementaci projektu byl využit programovací jazyk python. Tento jazyk jsem zvolil primárně z důvodu, že je velice vhodný v oblasti strojového učení. Obsahuje spousty knihoven, které například slouží pro práci s neuronovými sítěmi, pro počítání s rozsáhlými maticemi nebo pro vykreslování různých grafů, které se budou hodit během testování. Mezi knihovny, které jsem použil patří například numpy, matplotlib, keras či tensorflow.

8.1.2 Github

Pro verzování byl použit verzovací systém Git. Konkrétně Github. K přístupu na Git byl využit program Sourcetree. Git umožňuje verzovat jednotlivé verze projektu. Toho lze využít jako zálohu projektu nebo možnost vrátit projekt do starší verze bez toho abychom ztratili verzi aktuální.

8.2 Architektura

Jelikož měl projekt obsahovat více typů herních prostředí/deskových her a více typů hráčů neboli herních algoritmů, bylo potřeba vymyslet takovou strukturu projektu, aby nebyl problém kdykoliv přidat nového hráče nebo nový typ herního prostředí. Existuje jedna hlavní třída, ve které se skrývá logika hraní jedné čí více partií hry. Poté existuje třída, která zodpovídá za vykreslování grafů

ze statistik již dokončených zápasů. Nakonec jsou zde dvě rozhraní. Každé toto rozhraní obsahuje seznam funkcí, které musí třída implementovat, aby se mohla chovat jako hráč nebo herní prostřední. Funkce hlavní třídy, která zajišťuje chod hry, požaduje jako argumenty obě instance hráčů a instanci herního prostředí. Díky tomuto způsobu stačí pro výměnu hráče nebo herního prostřední pouze zaměnit tyto instance.

8.3 Herní prostředí

Nejdříve jsem se rozhodl implementovat herní prostřední, abych měl po vytvoření hráče možnost testovat jeho schopnosti. Aby třída mohla zastávat roli herního prostředí musela implementovat tyto funkce:

- reset - Funkce, která vykresluje herní plochu do původního stavu.
- get_game - Funkce, která vrátí o jaký typ herního prostředí se jedná.
- hash_value - Funkce, která vrátí momentální stav herní plochy jako haš.
- avaiable_moves - Funkce, která vrátí všechny možné pozice pro zahrání.
- who_won - Funkce, která vrátí, jaký hráč vyhrál momentální partii.
- other_player - Funkce, která vrátí hráče, který není momentálně na tahu. Využívá jí primárně minimax algoritmus.
- is_possible_move - Funkce, která jako argument bere pozici herního pole a vrací, jestli se jedna o validní tah pro zahrání.
- actual_state - Funkce, která vrátí aktuální stav herní plochy.
- states_count - Funkce, která vrátí velikost herní plochy.
- actions_count - Funkce, která vrátí počet možných akcí k zahrání.
- pygame_active - Funkce, která vrátí, jestli je momentálně aktivní PyGame rozhraní.
- random_empty_spot - Funkce, která vrátí náhodnou herní pozici validní pro tah.
- count_empty - Funkce, která počítá, kolik je volných pozic na herní ploše.
- move - Funkce, která slouží pro zahrání tahu hráče. Pozice tahu je poslána jako argument.
- step - Funkce, která slouží pro zahrání tahu hráče, který disponuje neuronovou sítí. Tato funkce oproti funkci move vrací také odměnu, která značí, jak byl tah dobrý.

- human_pygame_move - Funkce, která slouží pro zahrání tahu lidského hráče, pokud je aktivní PyGame rozhraní. Rozdíl je v tom, že pozice není poslána jako argument, ale funkce čeká na registraci kliku hráče na herní ploše.

- check_win - Funkce, která kontroluje, zda je hra u konce nebo stále probíhá.

- print_board_console - Tato funkce vypíše aktuální stav herní plochy do konzole.

8.3.1 TicTacToe

Jako prvního zástupce stolních her jsem si vybral velice známou hru TicTacToe. TicTacToe jsem si vybral z toho důvodů, že je tato hra při její základní variantě velice jednoduchá a lze u ní lehce poznat, zda je algoritmus či neuronová síť dobře naučená. Kouzlo je v tom, že ve variantě 3x3, kdy vyhrává ten, kdo první spojí tři políčka v řadě, lze zaručeně neprohrát. Jelikož se jedná o velmi malé herní pole, tak v případě, že oba hráči hrají naprosto bezchybně, by vždy měla nastat remíza.

Hra TicTacToe byla naimplementována tak, aby byla škálovatelná. Tudíž je možné nastavit velikost herního pole. Je to ovšem trochu omezené, vždy platí, že herní pole musí být čtverec a vítězný řetězec se rovná délce jedné strany. Takže pokud by herní pole bylo například 10x10 musí hráč pro vítězství spojit 10 znaků.

8.3.2 Connect4

Druhým zástupcem jsem zvolil Connect4. Hra byla vybrána z toho důvodu, že má prakticky stejný cíl jako první vybraná hra TicTacToe. Cílem hry je spojit 4 své znaky za sebou. Rozdíl je ovšem v tom, že nelze hrát znak na jaké políčko bychom chtěli. Při tahu si hráč vybere sloupec a jeho znak je vložen na nejspodnější řádek v tomto sloupci. Výběr hry byl podmíněn tím, že mě zajímalo, jestli se algoritmus, který se naučí hrát TicTacToe, dokáže bez problému naučit hru jinou, která má ovšem podobná pravidla. Hra se hraje na herním poli o rozloze 6x7.

8.3.3 PyGame Rozhraní

K možnosti hraní osobně proti některému z naimplementovaných algoritmů jsem se rozhodl vytvořit lehké grafické rozhraní, aby hráč nemusel psát například souřadnice do konzole. K implementaci jsem využil knihovnu PyGame. Postupoval jsem podobně jako u implementace herního prostředí. Vytvořil jsem si abstraktní třídu, ve které jsem implementoval funkce stejné pro všechny stolní hry. Jako například vykreslení obrazu nebo registraci tahu. Poté jsem definoval funkce, které už bude mít každé rozhraní jiné, například detekci kliku. V piškvorkách si hráč vybírá přímo políčko, do kterého chce hrát, na rozdíl od Connect4, kde si hráč vybírá pouze sloupec. Poté každé herní prostřední, které chce využívat toto grafické rozhraní, musí implementovat tuto třídu a její funkce.

8.4 Hráč

Aby třída mohla zastávat roli hráče, musela implementovat tyto funkce:

- move - Funkce, která zajišťuje tah hráče v herním prostředí
- final_result - Funkce, která se volá na konci jedné hry. Kromě toho, že zapisuje hráči výsledky, tak v některých algoritmech může dělat dodatečné funkce. Například u Q-Learningu tato funkce také spouští učící fázi.
- get_wins - Funkce vrátí počet vyhraných her.
- get_loss - Funkce vrátí počet prohraných her.
- get_draws - Funkce vrátí počet remíz.
- new_game - Funkce, která slouží jako restart před začátkem hry. Slouží například pro výměnu strany hráče.
- get_side - Vrátí za jakou stranu právě hráč hraje. Například u TicTacToe vrací, jestli právě hráč hraje za kolečka nebo křížky.
- ge_player_name - Funkce, která vrátí hráčovo jméno. Používá se v grafu, aby se vědělo, která data patří kterému hráči.

Instance hráče bohužel nelze jen tak jednoduše zaměnit, protože každý hráč potřebuje trochu jiné hodnoty pro inicializaci. Například lidskému hráči nebo hráči, který hraje náhodně stačí jako argument jméno. Ostatní algoritmy jako například minimax nebo DQN potřebují dodatečné argumenty. Minimax potřebuje při inicializaci vědět, jestli se má chovat deterministicky nebo nedeterministicky a hráč, který využívá neuronové sítě zase potřebuje vědět, kolik existuje stavů a akcí, aby mohl vytvořit vhodnou neuronovou síť.

8.5 Zobrazení statistik a průběh testování

Jak bylo zmíněno na začátku práce, během testování budeme různé algoritmy porovnávat mezi sebou. K tomuto zobrazení jsem se rozhodl využít knihovnu matplotlib, jejíž pomocí jsem si vytvořil třídu, která se starala o vykreslování statistik z jednotlivých testů.

Způsob testování probíhal následujícím způsobem. Před spuštěním testu bylo potřeba specifikovat kolik se bude hrát zápasů a kolik se v jednotlivém zápasu bude hrát her. Pro představu, pokud se stanovilo, že se bude hrát 100 zápasů a v každém zápasu se bude hrát 10 her, tak se celkově bude hrát 100*10 her. Déle bylo potřeba specifikovat, jaká hra se bude hrát a na konec také, jaké algoritmy proti sobě budou postaveny. Ze začátku jsem zvolil způsob testování celkem špatným

způsobem a vždy začínal jeden a ten samý hráč. To jsem postupem času změnil tak, aby se hráči po každé hře střídali. Níže můžete vidět ukázku průběhu jedné hry.

```
def play_game(board, p1, p2, i):
    p1.new_game(CROSS)
    p2.new_game(CIRCLE)
    board.reset()

    if i % 2 == 0:
        active = ActivePlayer.PLAYER_ONE
    else:
        active = ActivePlayer.PLAYER_TWO

    finished = False
    while not finished:
        if active == ActivePlayer.PLAYER_ONE:
            result, finished = p1.move(board)
            if finished:
                if result == GameResult.DRAW:
                    final_result = GameResult.DRAW
                else:
                    final_result = GameResult.CROSS_WIN
            active = ActivePlayer.PLAYER_TWO
        else:
            result, finished = p2.move(board)
            if finished:
                if result == GameResult.DRAW:
                    final_result = GameResult.DRAW
                else:
                    final_result = GameResult.CIRCLE_WIN
            active = ActivePlayer.PLAYER_ONE

    p1.final_result(final_result)
    p2.final_result(final_result)
    return final_result
```

Na začátku funkce můžeme vidět, že se vše vyresetuje do počátečního stavu. Poté se rozhodne jestli začíná první nebo druhý hráč, podle toho jestli je to sudá nebo lichá hra. Na konci se oběma

hráčům neboli algoritmům uloží, jak hra dopadla. Nakonec funkce vrátí výsledek o konci hry. Tento výsledek se používá ve funkci, která se stará o běh a statistiky jednoho zápasu.

```
def play_games(p1, p2, game, number_of_games):
    draw = 0
    cross_win = 0 #p1
    circle_win = 0 #p2

    for i in range(number_of_games):
        result = play_game(game, p1, p2,i)

        if result == GameResult.CROSS_WIN:
            cross_win += 1
        elif result == GameResult.CIRCLE_WIN:
            circle_win += 1
        else:
            draw += 1

    return cross_win, circle_win, draw
```

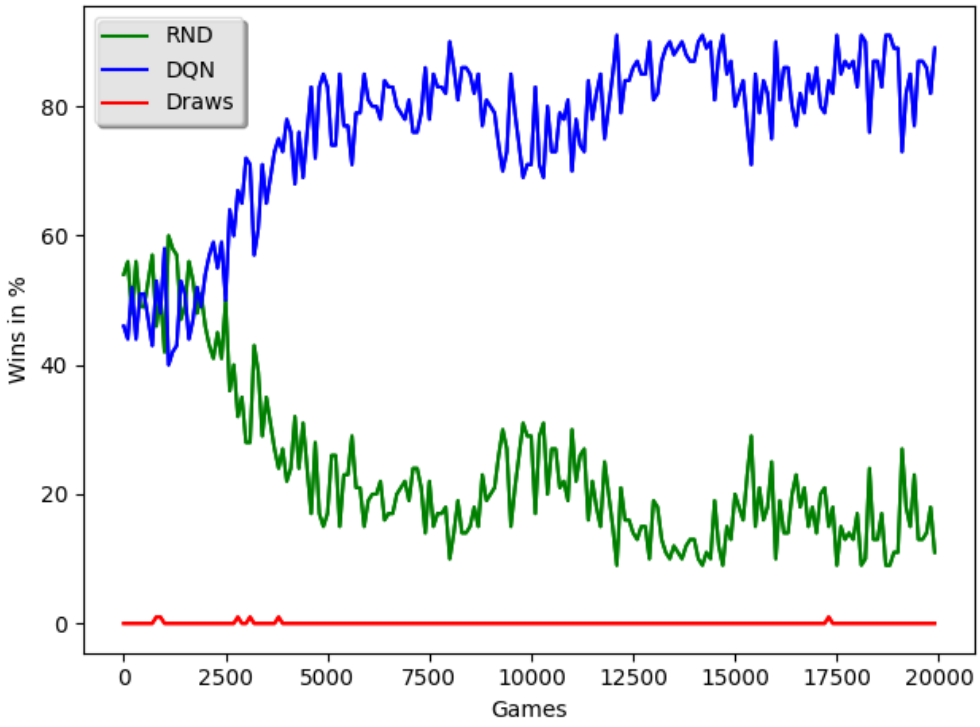
V této funkci tedy proběhnou všechny hry z jednoho zápasu, ve kterém se po každé hře připočítá výhra prvního hráče, výhra druhého hráče, nebo remíza. Po dokončení těchto her funkce vrátí statistiky o tom, jak celkově dopadl tento zápas.

```
def play_matches(p1, p2, game, number_of_matches, number_of_games, draw_graph):
    stats = Stats(p1.get_player_name(),p2.get_player_name())

    for i in range(number_of_matches):
        print("Match number {}".format(i))
        cross_win, circle_win, draw = play_games(p1, p2, game, number_of_games)
        stats.append_stats(cross_win,circle_win,draw,i,number_of_games)

    if draw_graph:
        graph = Graph()
        graph.draw_stat_graph(stats)
```

Poslední funkcí je funkce, která se stará o chod celého testování a shrnutí výsledku všech zápasů. Zde probíhají v cyklu jednotlivé zápasy a po každém je do objektu statistik, což je list, vložen záznam, který obsahuje počet výher každého hráče a remíz v jednotlivých zápasech. Také obsahuje počet celkových her pro následné vykreslení do grafu. Na konci se volá funkce pro vykreslení grafu,

ovšem pouze pokud funkci nastavíme, že chceme, aby se na konci graf vykreslil. Statistiky také počítají celkový počet výher, proher a remíz pro jednotlivé hráče. Tyto statistiky jsem si na konci nechal vypsat do konzole.

Je možné, že v průběhu práce budou na některých grafech trochu nepřesné popisky. Je to z toho důvodu, že jsem během testování objevoval, jak by se dal výpis statistik zlepšit a graf byl postupně zlepšován. Ve finální verzi grafu, lze na Y-ose viděl poměr procent výhry a na X-ose je časový průběh neboli kolik her již bylo odehráno.

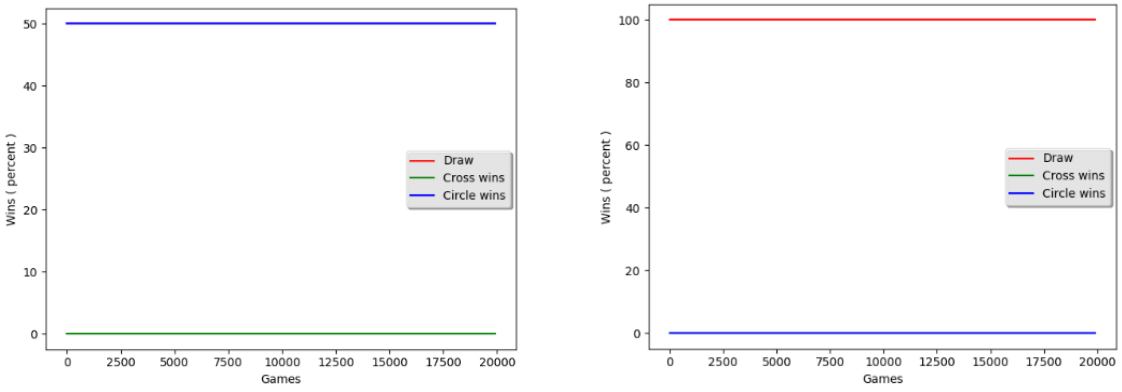


Obrázek 8.1: Ukázka grafu vytvořeného ze statistik testování

8.6 Minimax

Prvním implementovaným algoritmem, pokud nebudu počítat "algoritmus", který své tahy hraje náhodně, byl minimax. Jelikož se nejedná o algoritmus zpětnovazebního učení a jeho implementace je už vlastně nějakým způsobem dána, nebudu zde jeho implementaci podrobně rozepisovat. Zaměřím, se spíše na problémy, které mě během této implementace potkaly.

Jak jsem již zmiňoval, implementoval jsem dvě varianty tohoto algoritmu a to deterministickou a nedeterministickou. Při testování těchto dvou algoritmů mezi sebou nebo proti jiným (konkrétně random a Q-Learning), jsem narazil na jeden závažný problém. Jelikož tento algoritmus vždy vybere ten nejlepší možný tah, tak v každé hře by měl buď vyhrát nebo remízovat. Očekávané chování je takové, že pokud se proti sobě postaví dva tyto algoritmy, tak bude stoprocentní remíza. Tomu také tak bylo, dokud jsem později nezačal prohazovat začínajícího hráče. Při prvním testování po implementaci oné techniky střídaní hráčů jsem si všiml nezvyklého chování.



(a) Výsledné chování

(b) Očekávané chování

Obrázek 8.2: Porovnání chování algoritmus

Na obrázcích lze vidět, že místo toho, aby byla stoprocentní remíza, tak remíza nastala pouze z 50 procent a zbytek her vyhrál hráč, který začínal. Postupem času jsem zjistil, že problém byl v jednom vylepšení algoritmu, které jsem implementoval pro jeho zrychlení. Konkrétně se jednalo o transpoziční tabulku. Problém začínal nastávat až při střídání z toho důvodu, že tabulka začala vracet výsledky pro tah druhého hráče. Například pokud hráč začínal jako křížek, tak si algoritmus při první hře vytvořil transpoziční tabulku pro to, jaký tah použil při nějakém stavu. Tento tah se počítal jako maximalizace, jelikož začínal. Pokud se do toho stejného stavu dostal v momentu, kdy začínalo kolečko, tak se tento tah automaticky vrátil jako maximalizace i když měl být spočítaný jako minimalizace. To znamená, že se v celkovém rozhodnutí stala chyba a v momentě, kdy měl algoritmus vrátit nejlepší tah, tak vrátil tah nejhorší. Tento problém jsem nakonec vyřešil rozdělením transpoziční tabulky na dvě, jedna nesla hodnoty ohledně maximalizace a druhá ohledně minimalizace.

8.7 Q-Learning

Doufal jsem, že implementace Q-Learningu nebude tak složitá vzhledem k pár zkušenostem, které jsem s ním již měl. Konkrétně jsem v minulých semestrech implementoval Find the Cheese algoritmus, který používal právě Q-Learning. Tyto zkušenosti mi primárně pomohly v nastavování počátečních hodnot, které jsou velmi důležité pro správné naučení QLeaningu.

Funkcí, ze kterých se můj algoritmus skládá není moc. Je zde funkce, která vrátí Q-hodnoty pro všechny možné akce z určitého stavu. Tohoto využívá funkce, která vybere nejlepší akci z vybraných. Jako nejlepší akce je vybrána ta, která má největší hodnotu. Problémem je, že algoritmus může označit jako nejlepší akci pozici, na které už něco je, takže není možné tam zahrát tah. Z tohoto důvodu kontroluji, zda je vybraný tah tahem možným a pokud ne, nastavím jeho hodnotu na -1.

```
def get_q(self, board_hash, available_moves):
    if board_hash in self.q:
        qvals = self.q[board_hash]
    else:
        # AVAIABLE MOVES
        qvals = np.full(available_moves, self.q_init_val)
        self.q[board_hash] = qvals

    return qvals

def get_move(self, board):
    board_hash = board.hash_value()
    qvals = self.get_q(board_hash, board.available_moves())

    while True:
        m = np.argmax(qvals)
        if board.is_possible_move(m):
            return m
        else:
            qvals[m] = -1.0
```

Poté byla potřeba naimplementovat funkce pro samostatný pohyb. Tato funkce pouze zavolá na herní pole a předá mu souřadnice, na které má zahrát daný tah. Tento tah se uloží do takzvané historie tahů, kterou bude Q-Learning na konci hry využívat pro učení. Do historie se zaznamená zahašovaný stav herní plochy a tah, který je dle Q-Learningu v tomto stavu nejlepší. Po dokončení hry se vyhodnotí, který hráč vyhrál a celá historie se promítne jako materiál pro učení, podle kterého se upraví Q-table. Po naučení se historie vyprázdní, aby byla připravená pro následující partii.

```
def final_result(self, result):
    super().final_result(result)
    if (result == GameResult.CIRCLE_WIN and self.side == CIRCLE) or (
            result == GameResult.CROSS_WIN and self.side == CROSS):
        final_value = WIN
    elif (result == GameResult.CIRCLE_WIN and self.side == CROSS) or (
            result == GameResult.CROSS_WIN and self.side == CIRCLE):
        final_value = LOSS
    elif result == GameResult.DRAW:
        final_value = DRAW
    else:
        raise ValueError("Unexpected game result {}".format(result))

    self.move_history.reverse()
    next_max = -1.0

    for h in self.move_history:
        qvals = self.get_q(h[0], self.actions)
        if next_max < 0:
            qvals[h[1]] = final_value
        else:
            qvals[h[1]] = qvals[h[1]] * (
                        1.0 - self.learning_rate) + self.learning_rate * self.
                            value_discount * next_max
        next_max = max(qvals)
```

8.8 DQN

Implementace DQN algoritmu by se dala rozdělit do dvou části. V první části bude popsána samostatná implementace neuronové sítě a v druhé části poté hráč, který tuto neuronovou síť využívá.

8.8.1 Implementace neuronové sítě

K implementaci neuronové sítě jsem využil knihovnu keras, která mi usnadnila práci například tím, že nebylo potřeba implementovat back propagation. Samotná implementace se poté skládala primárně ze dvou kroků a to z vytvoření struktury sítě a funkce pro trénováni této sítě. Při vytváření struktury bylo potřeba stanovit 3 části. Vstupní vrstvu, skrytou vrstvu a výstupní vrstvu. To je možné udělat dvěma způsoby. První způsob byl vhodný pro základní sítě, ve kterých nebylo potřeba speciálních výpočtů. Bylo nutné vytvořit sekvenční model a poté do něho přidávat jednotlivé vrstvy tak, jak by měly jít za sebou. U druhého způsobu se nevytváří sekvenční model, ale vytváří se jednotlivé vrstvy, kterým se musí specifikovat, jaká jiná vrstva se bude chovat jako její vstup. Poté se vytvoří model stanovením vstupní a výstupní vrstvy. Model se nakonec musí zkompilovat, přičemž je možné nastavit hodnoty jako funkci ztráty nebo optimalizátor.

```
def create_model(self):
    model = Sequential()
    model.add(Dense(64, input_dim=self.state_size, activation="relu"))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(lr=LEARNING_RATE))

def create_model(self):
    input = Input(shape=self.state_size * 3)
    out = Dense(8, activation='relu')(input)
    value = Dense(4, activation='relu')(out)
    value = Dense(1, activation='relu')(value)
    advantage = Dense(4, activation='relu')(out)
    advantage = Dense(self.action_size, activation='relu')(advantage)
    advantage_mean = Lambda(lambda x: K.mean(x, axis=1))(advantage)
    advantage = Subtract()([advantage, advantage_mean])
    out = Add()([value, advantage])
    model = Model(inputs=input, outputs=out)
    model.compile(loss='mse', optimizer=Adam(lr=LEARNING_RATE))
```

Trénování funguje na podobné bázi jako učení u Q-Learningu. Všechny tahy, které algoritmus zahrál se ukládají do paměti. U neuronových sítí jsou ve formátu - počáteční stav, akce, odměna, konečný stav. Jakmile je těchto tahů v historii dostatek, může síť začít učit. Po každém tahu si síť vybere náhodné množství několika tahů z této historie a opakuje si je, aby se mohla lépe naučit, co byl špatný a co dobrý tah. Jelikož se to děje po každém tahu, tak je třeba trénovací funkci dobře optimalizovat. V první verzi trénovací funkce jsem projížděl jednotlivé tahy z historie jeden po druhém. U každého jsem predikoval jeho výslednou hodnotu a poté je všechny poslal do sítě. Tento způsob byl ovšem velmi pomalý. Podařilo se mi dosáhnout o dost rychlejší výsledků v momentu, kdy jsem u všech funkcí ohodnotil neboli predikoval jejich výslednou hodnotu ještě předtím, než jsem s nimi začal pracovat. Tudíž se predikce provedla pouze jednou pro X prvků, nežli X-krát podle počtů prvků.

8.8.2 Implementace DQN hráče

Po naimplementování neuronové sítě nastal čas implementovat hráče, který tuto síť bude využívat. Hráč se opět vytvořil za pomoci implementace rozhraní, které definovalo základní funkce, které hráč musel obsahovat. Jednou z funkcí, kterou tento hráč obsahoval navíc, byla funkce pro převedení herní plochy na korektní vstupní data pro neuronovou síť.

Nejpodstatnější byla implementace funkce pro pohyb, která úzce pracovala s neuronovou sítí. Tento hráč má navíc ještě atribut epsilon, který určuje s jakou pravděpodobností se má hráč rozhodnout, zda hrát náhodně nebo se spolehnout na neuronovou síť. Pokud se spolehnul na neuronovou síť, tak jí sdělil stav, ve kterém se právě nachází a nechal ji predikovat jakou akci má zahrát. Predikce funguje na stejné bázi jako Q-Learning, takže vrátí tabulku hodnot, kde největší hodnota znamená nejlepší tah. Opět se může stát, že nejlepší tah bude už v herním poli zabraný, takže je tyto nelegální tahy nutno odebrat. Zde nastal malý problém. Prvně jsem postupoval stejným způsobem jako u Q-Learningu a všechny nelegální tahy jsem nahradil hodnotou -1. Ovšem po delší době hrání se stalo to, že síť označila ostatní tahy tak špatně, protože vedly k prohře, že pole s hodnotou -1 se bralo jako nejlepší. To vedlo k výběru pozice, která už je zabraná a k následnému pádu programu. Snažil jsem se problém vyřešit zmenšováním této hodnoty na -10, -100 a tak dále, ale vždy byla pouze otázka času, než se tato nelegální pozice vybrala znovu. Nakonec jsem přišel s následujícím řešením. Namísto práce s negativními hodnotami jsem nelegální pozice nastavil na hodnotu *not a number* a tudíž nemohla nastat situace, že by tato pozice byla vybrána. Sice to trochu zkomplikovalo vybírání nejlepšího tahu, ale vyřešilo to chybné vybrání pozice

```
def get_best_legal_action(self, qtable, board):
    for index in range(0, len(qtable)):
        if not board.is_possible_move(index):
            qtable[index] = np.nan

    best_action = np.nan
    best_index = -1
    for index in range(0, len(qtable)):
        if qtable[index] is not None:
            if np.isnan(best_action):
                best_action = qtable[index]
                best_index = index
            else:
                if best_action < qtable[index]:
                    best_action = qtable[index]
                    best_index = index

    return best_index
```

8.9 Konvoluční sítě

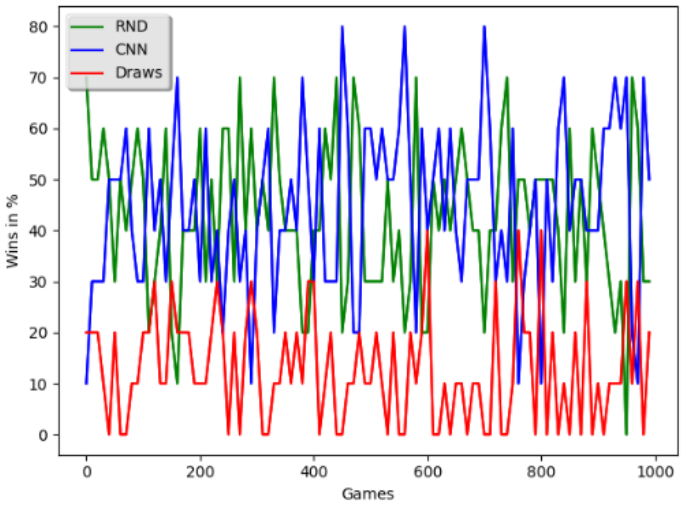
Poslední druh neuronových sítí, který jsem se rozhodl implementovat, byly sítě konvoluční. Jak již bylo zmíněno od DQN se primárně liší vstupními daty. Konvolučním sítím se dávají jako vstupní data většinou obrázky. Tyto obrázky se v procesu úpravy dat změní do formátu (n, x, y, channels). To znamená, že vstupní data, které jsem používal u DQN nelze použít u konvolučních sítí. Měl jsem dvě možnosti. První možností je si po každém tahu vyfotit herní plochu a tento obrázek poslat jako vstup. Druhou možností je si herní stav, který je uchován v poli, překonvertovat do správného tvaru. Vybral jsem si druhou možnost, protože představa, že bych musel po každém tahu nechávat fotit herní plochu, mi přišla z časového hlediska nemožná. Jelikož jsem vždy do sítě posílal pouze jeden vstup, neboli N bylo vždy rovno jedné, tak jsem vstupní data do sítě stanovil jako (x, y, channels).

```
def convert_to_cnn_input(self, state):
    state = np.reshape(state, (-1, 3))
    state = np.expand_dims(state, axis=-1)
    return state
```

Nejdříve jsem vstupní data, která byla v jednorozměrném poli, musel překonvertovat na pole dvojrozměrné. Poté jsem toto pole zvětšil o jednu dimenzi. Tyto data pro nás nesla nulovou hodnotu, ale je nutné je zde mít pro správné fungování. Tato upravená data se poté mohla předat konvoluční síti, které měla tento model.

```
input = Input(shape=(3,3,1))
output = Conv2D(64,kernel_size=(3,3), padding="same", activation="relu")(input
    )
output = Flatten()(output)
output = Dense(8, activation="relu")(output)
output = Dense(self.action_size, activation="linear")(output)

model = Model(inputs=input, outputs=output)
model.compile(loss='mse', optimizer=Adam(lr=LEARNING_RATE))
```

Při prvním testu, jsem vytvořil velice malý model, který obsahoval pouze jednu konvoluční vrstvu a poté pouze jednu dense vrstvu. Během implementování jsem začal uvažovat, zda jsou konvoluční sítě vůbec vhodné pro mou problematiku. Po nějakém bádání a přemýšlení jsem došel k tomu, že mé data jsou dost malých rozměrů pro to, aby je konvoluční sítě dobře využily. I přes to jsem zkusil síť nechat natrénovat abych viděl, jak to dopadne.



Obrázek 8.3: Statistika z testu konvolučních sítí

Na předešlém obrázku můžeme vidět, že se konvoluční sítě vůbec neučily, ale spíš hrály náhodně. Při konzultaci s vedoucím práce, jsem mu sdělil tyto informace a bylo mi potvrzeno, že konvoluční sítě opravdu nejsou vhodné pro mou problematiku, takže jsem od nich nakonec odstoupil.

Kapitola 9

Testování

Během celé implementace byly postupně testovány již kompletní algoritmy mezi sebou. Algoritmy jsem postupně upravoval na základě výsledků jednotlivých testů a tyto úpravy budou společně s detaily o testech popsány zde.

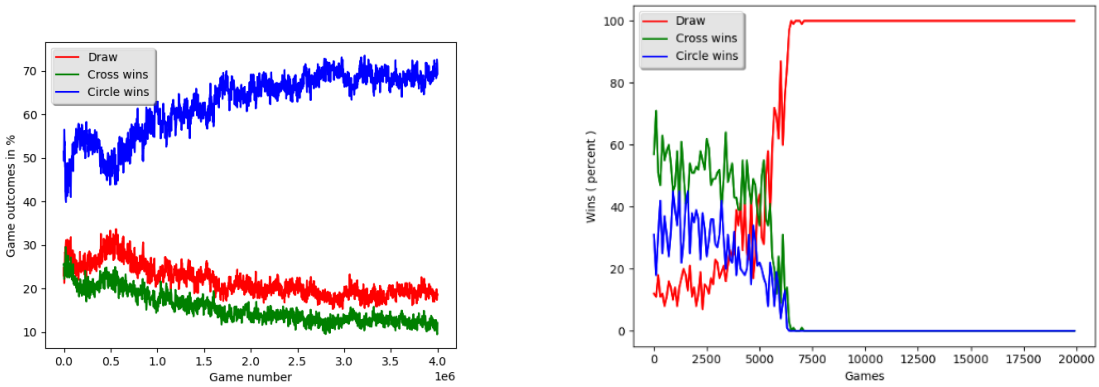
9.1 Q-Learning

Prvním adeptem na testování byl Q-Learning, který bylo možno v době testování postavit proti hráči, který hraje náhodně a proti minimaxu. Kvůli rychlosti výpočtu minimaxu pro hru Connect4, byly tyto první testy prováděny spíše na hře TicTacToe. Dalším důvodem, proč jsem se rozhodl testovat právě na hře TicTacToe je fakt, že v perfektní hře je výsledek vždy remíza. Díky tohoto faktu je možné za pomoci minimaxu zjistit, zda je algoritmus dobře naučený nebo ne.

Jelikož Q-Learning nedisponuje neuronovou sítí, proces učení není tak zdlouhavý. Hry, na kterých se bude tento algoritmus testovat, také nemají nějak obrovský počet stavů, takže si dokáže svou Q-table naplnit poměrně rychle. Půjde primárně o správné nastavení učícího faktoru, parametru gamma a správné odměny.

S odměnami byl trochu problém. Původní velikost odměny jsem zvolil stejně, jako je tomu u minimaxu. To znamená 1 bod za výhru, 0 bodů za remízu a -1 bod za prohru. Tyto odměny bohužel nevedly k úplně nejlepším výsledkům. Druhou variantou bylo dát negativní odměnu za prohru velice nízkou, něco jako -1000. To vedlo k ještě horším výsledkům. Nakonec jsem se dopátral k tomu, že best practice je mít odměnu mezi 1 a 0, takže konečné hodnoty odměny byly nastaveny takto.

- Výhra - 1
- Remíza - 0.5
- Prohra - 0



(a) proti Random

(b) proti Minimax (nonDet)

Obrázek 9.1: Testování QLearningu

Zde jsou názorné ukázky testů, podle kterých bylo zjištěno, že se Q-Learning naučil dobře. V grafu (a) jsou statistiky z her, ve kterých se Q-Learning učil proti hráči, který hraje náhodné pohyby. Tento test byl konkrétně proveden na hře TicTacTae o velikosti herního pole 4x4. Můžeme vypozorovat, že se Q-Learning postupem času zlepšoval. Pokud se mu nepodařilo vyhrát, snažil se to uhrát na remízu.

V druhém grafu (b) jsou statistiky z testu, kdy jsem proti sobě postavil Q-Learning a nedeterministický minimax. Výsledky dopadly podle očekávání a Q-Learning se po několik hrách naučil hrát stejně perfektně jako minimax, což vedlo k tomu, že všechny hry skončily remízou.

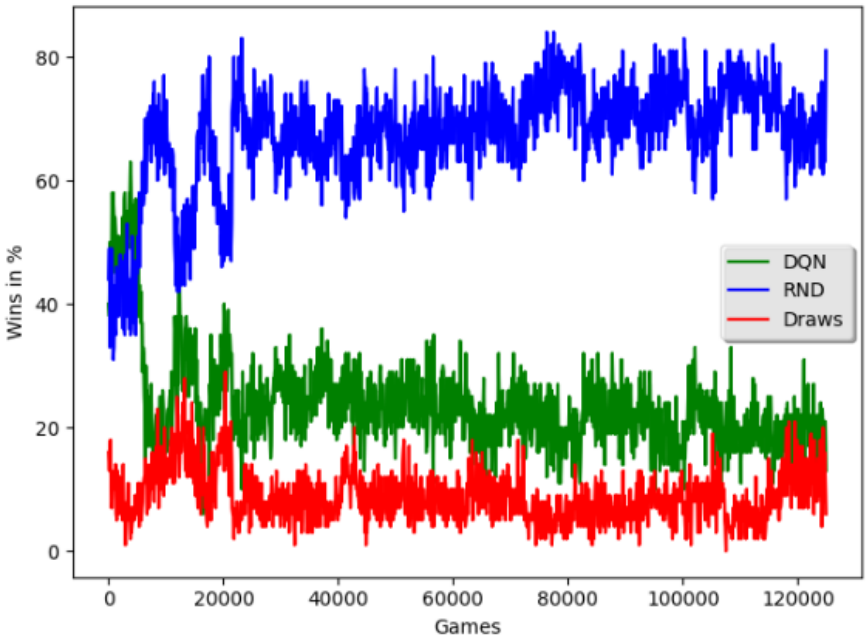
Tento algoritmus primárně slouží k rychlému naučení se nějakého úkolu většinou o menší velikosti, v našem případě hraní stolních her. Po těchto výsledcích jsem usoudil, že můžu Q-Learning považovat za naučený a posunout se dále, konkrétně k neuronovým sítím.

9.2 DQN

Tuto fázi testování popíšu ve dvou částech. V první části budu rozebírat testování na hře TicTacToe a poté testování na trochu komplexnějším Connect4. Je to hlavně z toho důvodu, že jsem u každé hry postupoval trochu jiným způsobem. Primárně proto, protože jsem nejprve testoval hru TicTacToe a až poté Connect4, tudíž jsem už měl už nějakou představu co upravovat nebo zlepšovat.

9.2.1 TicTacToe

Jak jsem zmínil, tak první hrou, na které jsem testoval DQN, byla hra TicTacToe. První test proběhl s obyčejným DQN proti hráči, který hraje náhodně. Jak můžeme vidět na následujícím obrázku, výsledky byly opravdu špatné. Neuronová síť se nedokázala naučit hrát ani proti obyčejnému algoritmu, který hrál na náhodné pozice.



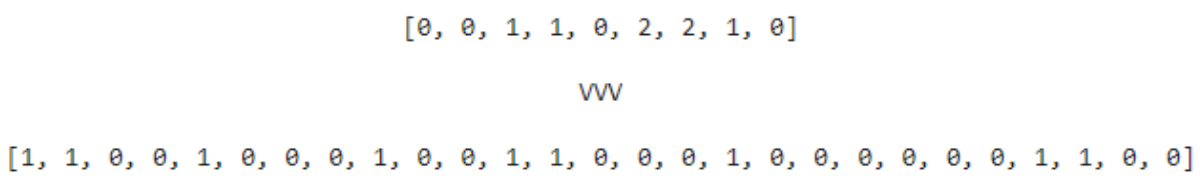
Obrázek 9.2: První test DQN vs Random

Faktorů, které toto špatné chování ovlivňovalo existuje více. Můžeme mezi ně zařadit například nevhodně zvolené odměny za určité tahy, velikost a celková struktura neuronové sítě. Výsledek také mohl ovlivnit tvar vstupních dat. Rozhodl jsem se postupně upravovat tyto jednotlivé parametry a sledovat zda se učení zlepšuje či nikoliv.

Přišlo mi, že hlavní problém bude v odměně, protože remízy se neodehrávaly skoro vůbec. To je u TicTacToe o velikosti herního pole 3x3 velice nepravděpodobné, zvláště když jeden z hráčů vybírá náhodná políčka. Odměny pro neuronovou síť byly nastaveny jako +1 za výhru a -1 za prohru. Takže jsem vlastně nedával žádnou váhu tomu, když se sítí podařilo dát dvě políčka do řady nebo remízovat. První úpravou tedy bylo dát nějakou váhu remíze a tahu samotnému. Remíze jsem se rozhodl dát odměnu polovinu výhry takže 0.5. U tahu bylo odměnu vymyslet náročnější. Ze začátku jsem nastavil nějakou nízkou kladnou hodnotu, protože tah, po kterém neprohraje by se dal brát jako krok k vítězství či remíze. Po troše testování mi došlo, že v tomto typu her neexistuje jediná situace, kdy prohrajete po právě svém zahraném tahu, takže mi přišlo nelogické dávat kladnou odměnu za tah, který může skončit dalším tahem oponenta. Zkusil jsem tedy za každý zahraný tah dávat negativní odměnu. Toto nakonec sítí dost pomohlo k tomu aby hrála lépe.

Další negativní účinky mohl mít na svědomí formát vstupních dat. Na začátku jsem se rozhodoval mezi dvourozměrným nebo jednorozměrným polem. Rozhodl jsem se pro vstup použít pole jednorozměrné, protože je pro neuronovou síť lehčí ke zpracování. Později jsem vyzkoušel ještě jiný vstupní formát, který se nakonec ukázal jako lepší volba. Starý vstup byl roven velikosti herního pole. Takže pro 3x3 pole vzniklo jednorozměrné pole o velikosti 9, ve kterém se označovala prázdná

pozice jako 0, křížek jako 1 a kolečko jako 2. Nová vstupní data jsem předělal tak, aby obsahovala pouze 0 a 1. Konkrétně se jednalo o tři pole zařazená za sebou, kde jedno pole značilo prázdná políčka, druhé křížky a třetí kolečka. Myslím, že bude lehčí tento vstup ukázat obrázkem.



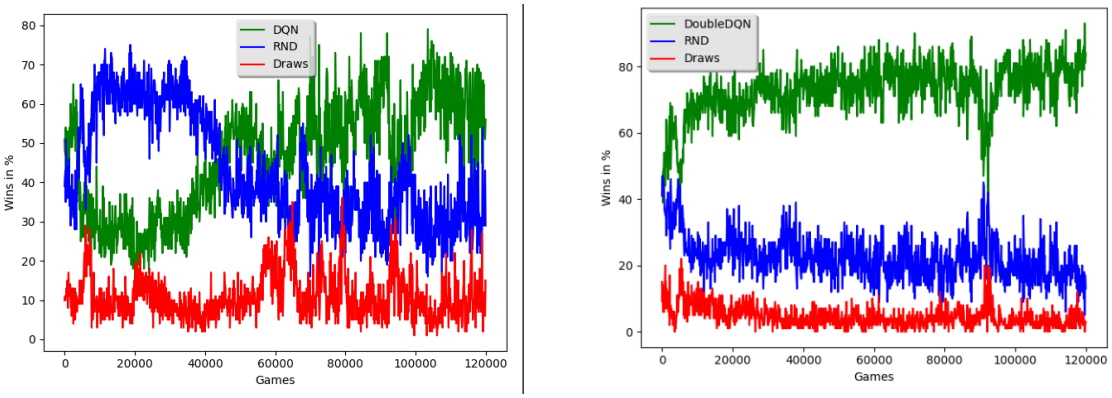
Obrázek 9.3: Přepočet vstupních parametrů

Poslední věc, kterou jsem mohl ovlivnit, je samostatná struktura sítě. Po dobu testování jsem vyzkoušel hodně kombinací. Primárně jsem se zaměřil na počet skrytých vrstev a počet neuronů v jednotlivých vrstvách. Některé z kombinací, které jsem vyzkoušel:

- 1. Input -> Hidden(32 neuronů) -> Hidden(16 neuronů) -> Hidden(8 neuronů) -> Output
- 2. Input -> Hidden(32 neuronů) -> Hidden(16 neuronů) -> Output
- 3. Input -> Hidden(27 neuronů) -> Hidden(27 neuronů) -> Output
- 4. Input -> Hidden(16 neuronů) -> Hidden(16 neuronů) -> Output
- 5. Input -> Hidden(8 neuronů) -> Hidden(4 neurony) -> Output
- 6. Input -> Hidden(4 neurony) -> Hidden(2 neurony) -> Output¨
- 7. Input -> Hidden(32 neuronů) -> Output

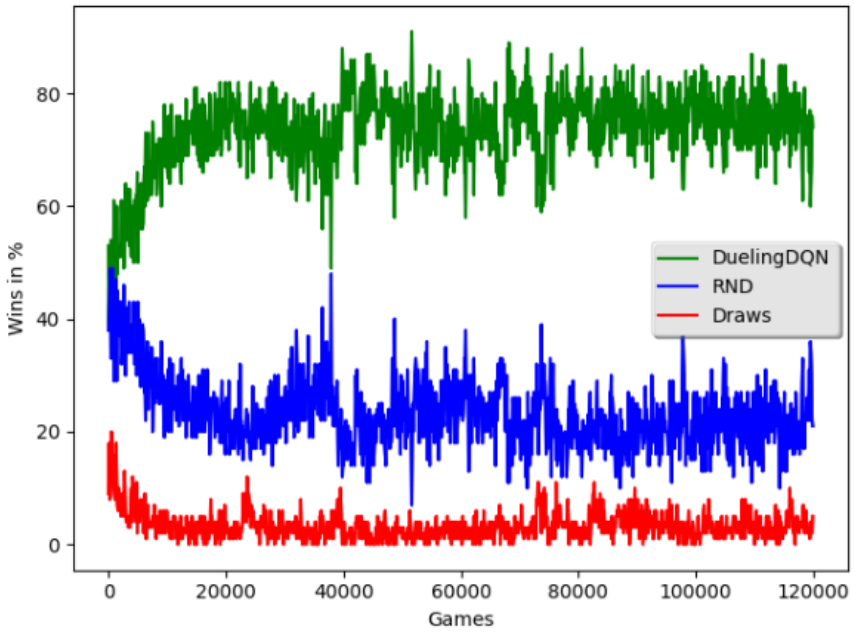
Jelikož se jednalo o poměrně malý problém, tak jsem se rozhodl síť zmenšovat s myšlenkou, že není potřeba tolik výpočtů pro správnou předpověď. Toto se nakonec ukázalo jako správná cesta a nejlepších výsledků jsem dosáhl s neuronovou sítí číslo 5.

Když jsem se sítí DQN dosáhl docela uspokojivých výsledků, rozhodl jsem se provést test mezi třemi podobami tohoto učení. Všechny podoby měly stejné podmínky. Hrály proti stejnému hráči, měly stejnou strukturu sítě, stejný formát vstupních dat a také stejné velikosti odměn. Jednalo o DQN, Double DQN a Dueling DQN.



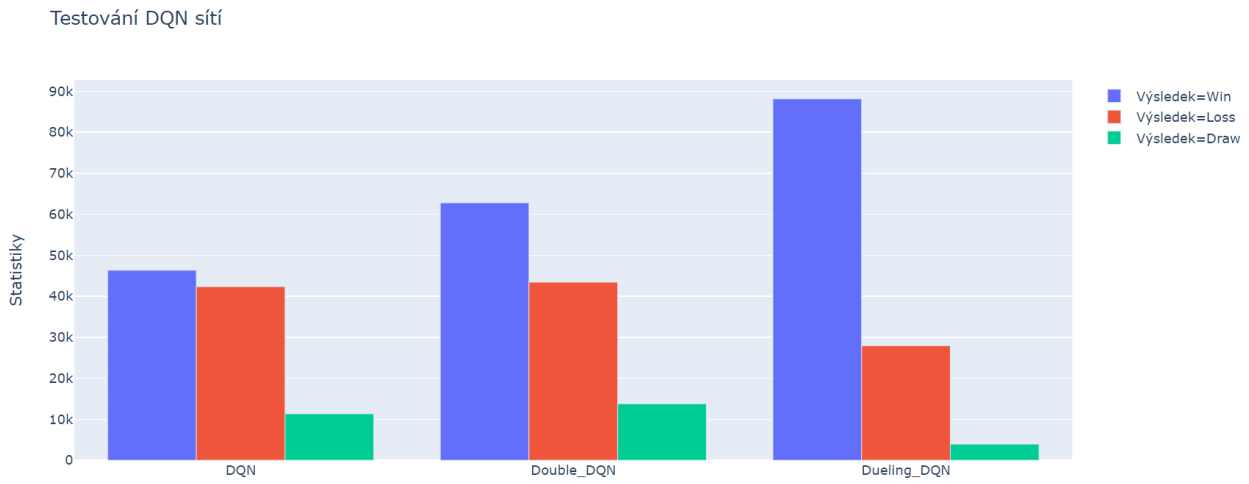
(a) DQN (b) Double DQN

Obrázek 9.4: DQN a DoubleDQN



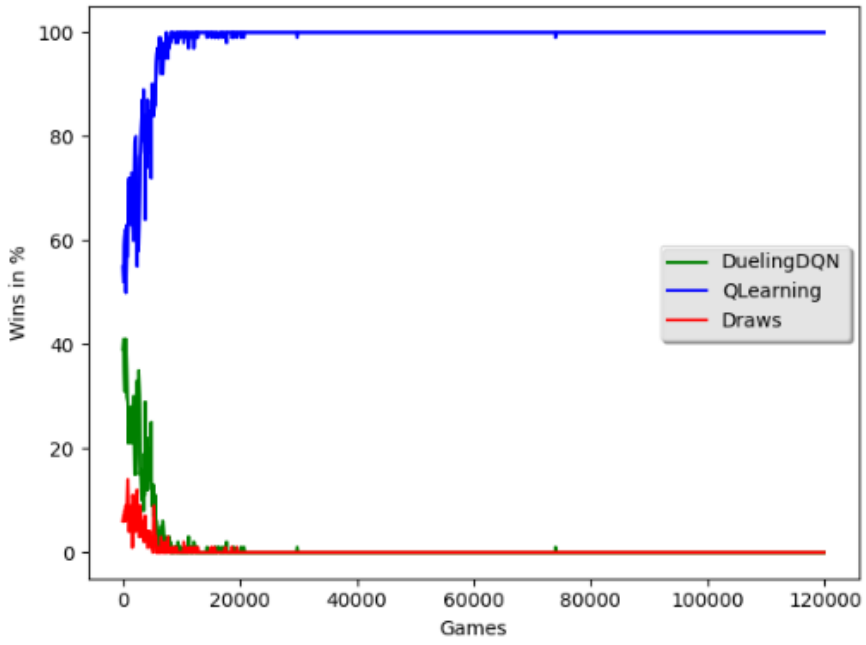
Obrázek 9.5: Dueling DQN

Výsledky dopadly dle očekávání a DQN se projevila jako nejhorší. Jeho mírné vylepšení Double DQN mělo výsledky o něco lepší, ale stále to nebylo dostačující. Dueling DQN se ukázala jako nejlepší, dokonce se již začala podobat chováním Q-Learningu, tak jak by se od DQN očekávalo.



Obrázek 9.6: Porovnání jednotlivých sítí

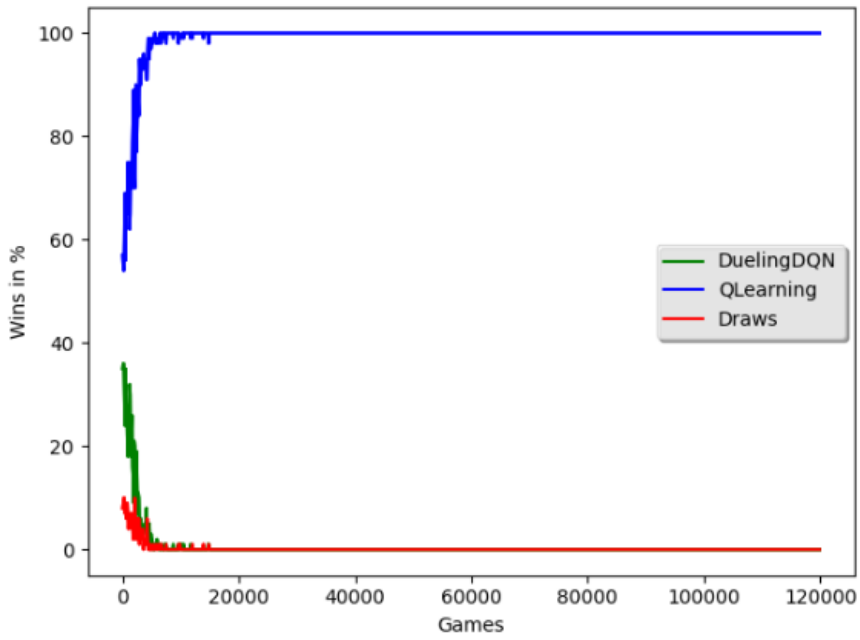
Po testování proti náhodnému hraní bylo na čase vyzkoušet, jak si DQN poradí se svým "originálem", Q-Learningem. První test spočíval v postavení nenaučené sítě proti nenaučenému Q-Learningu. Chtěl jsem zjistit, zda se dokáží učit navzájem. Test dopadl nad má očekávání, ovšem ne v pozitivním dopadu. Q-Learning se mnohonásobně rychleji naučil taktiku, jak vyhrát a poté DQN doslova zničil.



Obrázek 9.7: Dueling DQN vs QLearning před naučením

V dalším testu jsem chtěl proti nenaučenému Q-Learningu postavit již nějakým způsobem naučenou neuronovou síť. Ovšem ne takovou, která se učila hrát pouze proti někomu, kdo hrál náhodně. Rozhodl jsem se proto nejprve postavit proti sobě jednotlivé neuronové sítě, aby se učily proti sobě, a nakonec vybrat tu, které měla nejlepší výsledky vůči ostatním. V prvním kole se proti sobě postavila každá síť proti každé a všechny sítě začínaly v počátečním nenaučeném stavu. V druhém kole se proti sobě opět postavila každá proti každé, ale s nasbíranými zkušenostmi z prvního kola. Z každého zápasu jsem si nechal vytvořit grafy, abych dle nich mohl vybrat, která síť se dokázala nejlépe naučit. Jak jsem předpokládal, nejlépe dopadla Dueling DQN, která ke konci měla stoprocentní pravděpodobnost výhry.

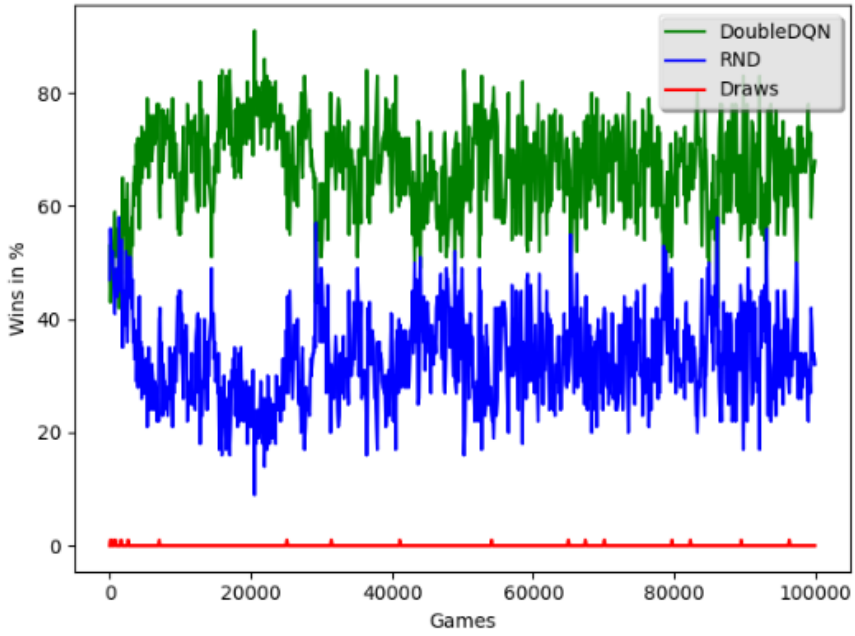
Jakmile jsem měl naučenou síť, přišel čas na odvetu mezi neuronovou sítí a Q-Learningem. Jak bylo v plánu, postavil jsem proti sobě naučenou Dueling DQN a nenaučený Q-Learning. Výsledek se bohužel skoro vůbec nezměnil a Q-Learning opět převálcoval neuronovou síť. Myslím si, že jedním z hlavní důvodů je, že TicTacToe je hra, která má pro Q-Learning poměrně malý počet stavů, takže se dokáže naučit hrát perfektně za velmi krátkou dobu.



Obrázek 9.8: Dueling DQN vs Q-Learning po naučení

9.2.2 Connect4

Testování hry Connect4 bylo lépe organizované, protože jsem z předchozích zkušeností věděl, na co se zaměřovat. První test opět proběhl mezi základní DQN a hráčem, který hraje náhodně. Test nedopadl tak zle jako tomu bylo u TicTacToe, ovšem nedopadl také nejlépe.

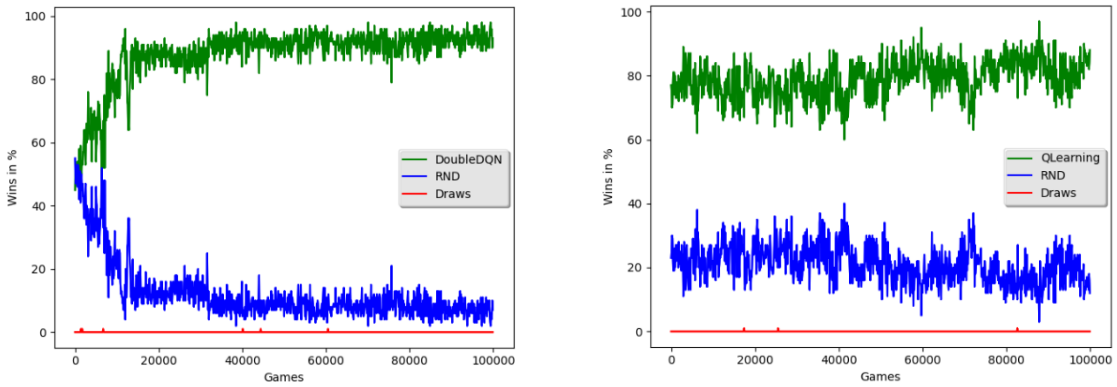


Obrázek 9.9: Dueling DQN vs QLearning po naučení

U této hry jsem se primárně zaměřil na strukturu neuronové sítě. Při prvním testu jsem nechal stejnou síť jako TicTacToe, ovšem ta nestačila. Usoudil jsem tedy, že síť musí být větší. Zde jsou opět některé sítě, které jsem vyzkoušel:

- 1. Input -> Hidden(4 neurony) -> Hidden(2 neurony) -> Output
- 2. Input -> Hidden(32 neuronů) -> Hidden(16 neuronů) -> Output
- 3. Input -> Hidden(64 neuronů) -> Hidden(32 neuronů) -> Output
- 4. Input -> Hidden(32 neuronů) -> Hidden(16 neuronů) -> Hidden(8 neuronů) -> Output
- 5. Input -> Hidden(64 neuronů) -> Hidden(32 neuronů) -> Hidden(16 neuronů) -> Output

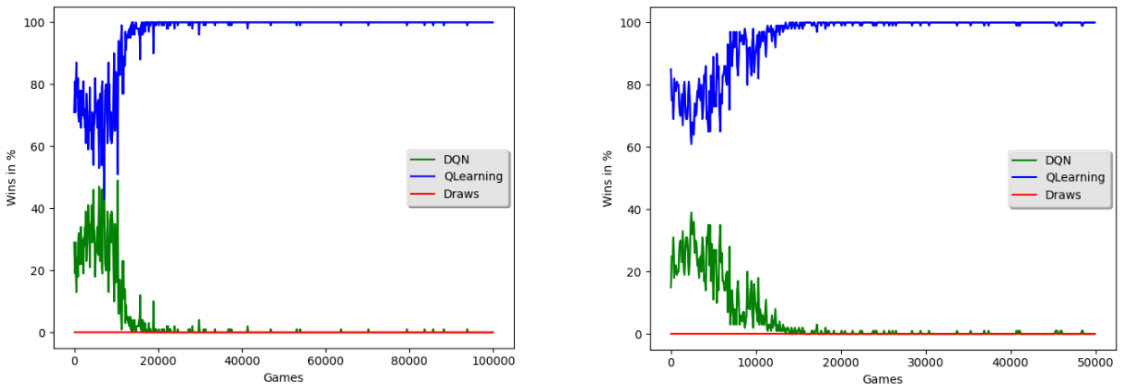
U sítě číslo 5. jsem dosáhl očekávaných výsledků, takže jsem nadále nezkoušel neuronovou síť měnit. Pokud by se mi nepodařilo dosáhnout těchto výsledků, tak další změnou, kterou jsem chtěl provést, byla změna vstupních dat stejným způsobem, jako tomu bylo u TicTacToe. Takže je převést pouze na 0 a 1. Zde je pro porovnání výstup mé neuronové sítě a výstup Q-Learningu proti stejnému oponentu.



(a) Double DQN (b) Q-Learning

Obrázek 9.10: Testování DQN oproti Q-Learningu

Dalším testem mělo být opět porovnání, zdali se DQN dokáže vyrovnat svému modelu DQN. Jelikož první test měl opět katastrofální výsledky, rozhodl jsem se postupovat stejně jako u TicTacToe. To znamená naučit jednotlivé sítě mezi sebou a poté vybrat tu nejlepší z nich. Ovšem ani v druhém testu, ve kterém byla neuronová síť naučená, jsem nedosáhl pozitivních výsledků a Q-Learning byl opět diametrálně lepší.



(a) Před naučením (b) Po naučení

Obrázek 9.11: DQN vs Q-Learning

Kapitola 10

Závěr

Během vypracování diplomové práce se mi podařilo nabít zkušenosti v oblasti strojového učení a tyto zkušenosti poté převést do praxe. Úspěšně se mi podařilo implementovat algoritmy, které spadají pod zpětnovazební učení. Sem spadá také implementace neuronových sítí, které jsou schopné se naučit hrát vybrané stolní hry. K dosažení těchto výsledků mi napomohly předměty jako Nekonvenční Algoritmy a Výpočty, kde jsem získal základy ohledně neuronových sítích, na kterých jsem poté mohl stavět a Metody analýzy dat, díky kterým jsem byl schopen správně zaznamenat výsledky jednotlivých testů.

Jelikož jsem tuto práci psal takovým způsobem, aby se dala jednoduše rozšiřovat, tak není problém na práci dále pokračovat. Do projektu lze vcelku lehce přidávat další věci, takže přidání nové hry nebo implementace nového algoritmu a následné integrace do testování, by neměl být žádný problém.

Během testování různých algoritmů se prokázalo, že pro hry s menší komplexností, jako je zrovna TicTacToe nebo Connect4, je pravděpodobně lepší využít Q-Learning. Nejvíce mě k tomu přesvědčily pozdější testy, ve kterých se Q-Learning prokázal svým rychlým učením. Jelikož hry mají poměrně malý počet možných stavů, tak se je Q-Learning dokáže naučit za velmi krátkou dobu, takže netrvá dlouho, než začne hrát perfektně. Ovšem u her s větší komplexností, jako jsou například šachy, by určitě byla správná volba zvolit neuronové sítě.

Práce jako taková mi velmi pomohla rozvinout mé programátorské myšlení, jelikož to bylo úplně něco jiného, než na co jsem byl doposud zvyklý. Člověk musí přemýšlet trochu jinak, než při implementaci například nějakého informačního systému. Problematika strojového učení je velmi zajímavá a určitě bych toto téma mohl doporučit budoucím studentům.

Literatura

1. HAO, Karen. *What is machine learning?* [Online] [cit. 2021-03-21]. Dostupné z: https://www.technologyreview.com/2018/11/17/103781/what-is-machine-learning-we-drew-you-another-flowchart/.
2. KHALED, Ahmed Yahya. *Cluster Analysis With Iris Data Set* [online] [cit. 2021-04-10]. Dostupné z: https://medium.com/swlh/cluster-analysis-with-iris-data-set-a7c4dd5f5d0.
3. *Regression Analysis in Machine learning* [online] [cit. 2021-04-10]. Dostupné z: https://www.javatpoint.com/regression-analysis-in-machine-learning#:~:text=Regression.
4. LEE, Dan. *Reinforcement Learning, Part 1: A Brief Introduction* [online] [cit. 2021-04-10]. Dostupné z: https://medium.com/ai-theory-practice-business/reinforcement-learning-part-1-a-brief-introduction-a53a849771cf.
5. *Reinforcement Learning : Markov-Decision Process* [online] [cit. 2021-03-21]. Dostupné z: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da.
6. SURMA, Greg. *Cartpole - Introduction to Reinforcement Learning* [online] [cit. 2021-03-21]. Dostupné z: https://gsurma.medium.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288.
7. JAGTAB, Rohan. *Understanding Markov Decision Process (MDP)* [online] [cit. 2021-03-21]. Dostupné z: https://towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150.
8. OSIŃSKI, Błażej; BUDEK, Konrad. *What is reinforcement learning? The complete guide* [online] [cit. 2021-04-10]. Dostupné z: https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/.
9. ARADHYA, Akshay L. *Minimax Algorithm in Game Theory* [online] [cit. 2021-03-21]. Dostupné z: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/.

10. ARADHYA, Akshay L. *Minimax Algorithm in Game Theory* [online] [cit. 2021-03-21]. Dostupné z: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/.

11. VIOLANTE, Andre. *Simple Reinforcement Learning: Q-learning* [online] [cit. 2021-03-21]. Dostupné z: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56.

12. *Taking into account symmetry, how many possible games of tic-tac-toe are there?* [Online] [cit. 2021-03-21]. Dostupné z: https://math.stackexchange.com/questions/2476469/taking-into-account-symmetry-how-many-possible-games-of-tic-tac-toe-are-there.

13. JAMES, Mike. *Number Of Legal Go Positions Finally Worked Out* [online] [cit. 2021-03-21]. Dostupné z: https://www.i-programmer.info/news/112-theory/9384-number-of-legal-go-positions-finally-worked-out.html.

14. *Introduction to Neural Networks and Deep Learning* [online] [cit. 2021-03-21]. Dostupné z: https://medium.com/@societyofai/introduction-to-neural-networks-and-deep-learning-6da681f14e6.

15. GUDIKANDULA, Purnasai. *A Beginner Intro to Neural Networks* [online] [cit. 2021-03-21]. Dostupné z: https://purnasaigudikandula.medium.com/a-beginner-intro-to-neural-networks-543267bda3c8.

16. MILAN, Data Science. *Time Series Classification with Deep Learning* [online] [cit. 2021-04-10]. Dostupné z: https://datasciencemilan.medium.com/time-series-classification-with-deep-learning-205db7b47e1e.

17. *Weights and Biases* [online] [cit. 2021-03-21]. Dostupné z: https://docs.paperspace.com/machine-learning/wiki/weights-and-biases.

18. MOUJAHID, Adil. *A Practical Introduction to Deep Learning with Caffe and Python* [online] [cit. 2021-04-10]. Dostupné z: http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/.

19. AUSIN, Markel Sanz. *Introduction to Reinforcement Learning. Part 3: Q-Learning with Neural Networks, Algorithm DQN* [online] [cit. 2021-03-21]. Dostupné z: https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e22ee928ecd.

20. TORRES, Jordi. *Deep Q-Network (DQN)-II* [online] [cit. 2021-03-21]. Dostupné z: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c.

21. AUSIN, Markel Sanz. *Introduction to Reinforcement Learning. Part 4: Double DQN and Dueling DQN* [online] [cit. 2021-03-21]. Dostupné z: `https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-4-double-dqn-and-dueling-dqn-b349c9a61ea1`.

22. MISHRA, Mayank. *Convolutional Neural Networks, Explained* [online] [cit. 2021-03-21]. Dostupné z: `https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939`.

23. SAHA, Sumit. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way* [online] [cit. 2021-03-21]. Dostupné z: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`.